A Comparison of Transformation Tools Based on the Transformation Tool Contest

Edgar Jakumeit[a,*], Sebastian Buchwald[b], Dennis Wagelaar[c], Li Dan[d], Ábel Hegedüs[e], Markus Herrmannsdörfer[f], Tassilo Horn[g], Elina Kalnina[h], Christian Krause[i], Kevin Lano[j], Markus Lepper[k], Arend Rensink[l], Louis Rose[m], Sebastian Wätzoldt[n], Steffen Mazanek[o]

[a]?
[b]*Karlsruher Institut für Technologie, Germany*
[c]?
[d]?
[e]*Budapest University of Technology and Economics, Hungary*
[f]*Institut für Informatik, Technische Universität München, Germany*
[g]?
[h]?
[i]*Hasso-Plattner-Institut, Universität Potsdam, Germany*
[j]?
[k]?
[l]?
[m]?
[n]*Hasso-Plattner-Institut, Universität Potsdam, Germany*
[o]?

Abstract

Model transformation is one of the key tasks in model-driven engineering and relies on the efficient matching and modification of graph-based, high-level data structures. Moreover, graph transformation has been shown to be a suitable modeling language for distributed and mobile systems. Over the last years, a wide range of model and graph transformation tools have been developed – each of them with its own particular strengths and typical application domains. In this paper, we give an overview and a comparison of the graph and model transformation tools that participated at the Transformation Tool Contest 2011. This overview is based on the solutions to a

[*]Corresponding author
Email address: `eja@ipd.info.uni-karlsruhe.de` (Edgar Jakumeit)

Hello World challenge, which was designed with instructiveness for beginners in mind. The aim of this comparison is to aid the users with a transformation task at hand in choosing the right tool for their needs. All solutions referenced in this article provide a SHARE demo, which allows the reader to test the tools online and to support the open peer-review process.

Keywords: tool comparison, hello world, transformation tool contest, SHARE, graph rewriting, model transformation

1. Introduction

The work in this article is part of the special issue on Experimental Software Engineering in the Cloud of the journal Science of Computer Programming, which collates articles that compare software engineering tools in a pragmatic manner, for example by having tool developers solve a shared case study. The focus of this special issue is on reproducibility and accessibility, to be satisfied with virtual machine demos.

1.1. Background and Motivation

The Transformation Tool Contest [78] held 2011 in Zurich, Switzerland, was a research workshop where vendors of transformation tools from the worlds of model transformation and graph rewriting gathered to compare their tools along a number of selected case studies. At this workshop, SHARE [20] virtual machines were employed to support the peer-review process.

The compared solutions were implemented with complex software toolkits, often requiring a substantial amount of time and knowledge to install and set up. The SHARE cloud allowed all participants to start up the pre-installed software environment of their competitors including their solutions with just one click, drastically lowering the barrier for comparison and evaluation, thus satisfying a central requirement of the call for contributions.

While being meant originally for peer-based evaluation, the SHARE images of this scientific workshop were kept available for further empirical evaluation by prospective tool users from industry, who can explore the effect of inputs and parameters better suited to their needs. This is highly important, as tool comparisons are not a goal on their own: while they should help the tool vendors to improve their tools by learning from the strengths and weaknesses of the other tools, they should especially help potential users in choosing a tool, aided by an understanding of the consequences of the different approaches.

Based on the comparisons of the Transformation Tool Contest, we focus in the following on the usability of the tools for certain tasks. The primary goal of this article is to help software engineers that are facing a specifc model transformation or graph rewriting problem in choosing a tool that is well-suited to their needs. Given such a statement, the first question that arises is: What is *a model transformation* or *a graph rewrite problem*? And the second question is: What does *well-suited to someone's needs* mean?

1.2. Problem domain

In a nutshell, we are confronted with a graph rewrite or model transformation problem if the most adequate representation of the data at hand is a mesh of objects, and we need to change its structure.

A *model* is an abstract representation of a system, which captures exactly the characteristics of the system that are of interest to the model designer [68]; a complete system may be covered by models at various abstraction levels described from different points of view. Models are typically built from entities and relationships between these entities, forming a network of interrelated objects. They conform to a metamodel (built from entities and relationships) to be defined by the designer while modeling. The metamodel in turn conforms to a fixed metametamodel.

The mathematical concept of a *graph* is very similar to the model notion: Here the entities are called *nodes*, and the relationships *edges*, forming a network of interconnected nodes. In the rest of this paper, we regard both concepts to be equivalent and use them interchangeably. A programmer is facing a graph- or model-based problem if the domain of interest requires a representation of the system to be modeled as network of objects. Problems that can be adequately modeled with scalar values or lists do not benefit from any of the tools introduced in this article.

One domain in which the introduced tools are helpful is model-driven development according to the Model-Driven Architecture vision [58] of the OMG. In this approach, the central artifacts in the software development process are models. Software is developed by defining metamodels and implementing transformations between them, which finally yield an executable model or program code. In this context, we are typically confronted with *transformation* problems, which require translating higher-level program representations into lower-level program representations. Other domains in which this kind of tools are helpful include, among others, mechanical engineering [25], computer linguistics [3], and protocol verification [63]. In these areas, we are typically confronted with *rewrite* problems, which require modifying one graph by sequence of transformation steps. In each of these steps, a graph pattern needs to be matched and replaced by another pattern until a goal state is reached.

1.3. Solution domain

Given a problem as described above, what are the possibilities in solving it? One could either

1. code it by hand in a high-level programming language of one's own choice, or

2. use one of the transformation tools introduced in this article.

The question that now naturally arises is: What are the benefits of using one of the tools compared to manual coding? The direct, simple answer is *reuse*. A tool offers a proven and tested implementation, which was already debugged and is ready to be used. Reusing an existing tool allows a developer to achieve his goals quicker and with less effort.

The languages offered by the tools are typically of a much higher *expressiveness* for tasks of the domains of model transformation or graph rewriting than general-purpose programming languages. This holds because subproblems of the domain, which required explicit imperative code before were solved and made *declarative* by the tool vendors through implementing a code generator or a runtime library. A code generator emits the previously needed imperative code upon visiting a declarative specification standing for it; a runtime library executes the code that the developer usually would have to write on his own.

Solutions specified in these special-purpose languages are *concise* compared to solutions coded in general-purpose programming languages. This leads to a decrease in the development and maintenance costs as the developer needs to write less code during the development and read or change less code during maintenance. Additionally, graph and model transformation tools may offer a more user-friendly graphical access to models and model transformations. Often, these tools offer a *visual* specification style and to highlight patterns matching in a graph and effects of rewriting steps.

A common problem of reuse is: Are the available features really the ones that are needed? Using a caliper when a hammer is needed would not offer the expected benefits as the effort invested into tool building could not be reused. Thus, besides the question of how much code is offered for reuse by the tool, the more important question is how much of this code can be exploited for the task at hand. A developer must *choose* wisely according to his needs.

To support the prospective user in his choice, the tools presented in this article are introduced alongside an illustrative subtask of a *Hello World* task, together with a discussion of their particular strengths and weaknesses. Furthermore, they are compared in detail alongside feature matrices motivated by typical questions a user might ask. Some may reveal a tool is not expressive enough at all to tackle the task at hand (or not extensible, which would allow to use it nevertheless if the other features were compelling). Others may give shades of gray regarding expressiveness, revealing if a given task can be solved directly in a declarative and concise way, or only with a more imperative combination of certain elements, hampering readability and maintainability. Tools supporting more than needed have the advantage that the chances are higher that they stay a good choice in case the requirements change after the project start. On the other hand, they may unnecessarily burden the user with their more heavyweight constructs for simple tasks.

1.4. Potential benefits and drawbacks

We now discuss in more detail the potetial benefits and drawbacks of using a transformation tool instead of a manually implemented solution.

Graph and Model. In the object-oriented paradigm, node types would be typically implemented by classes, nodes by objects, and edges by references stored in the source object, pointing to the target object. In contrast, a transformation tool or a special-purpose language may offer:

- A more concise specification. For the easiest manually coded solution the benefits would not be compelling. However, optimizing the implementation to achieve a high performance (especially for pattern matching) is technically challenging.
- A more elaborate model, e.g., featuring attributed edges.
- Run-time adaptability of the metamodel.

- Reuse of importer/exporter code.
- A graph viewer that can be used to visually inspect the model, instead of being forced to chase chains of references in the debugger of the programming language used for a manual implementation.
- A graphical metamodel editor.
- An explicit model interface that abstracts from the underlying modeling, enabling the reuse of computations on different modeling technologies.

Rewriting and Transformation. The most simple manually implemented solution would navigate the object graph with loops, explicitly manipulating the processing state. In contrast, a transformation tool or a special-purpose language may offer:

- A more concise specification, with e.g., navigational expressions.
- A declarative specification with rules, which specify the graph patterns to be matched and modified.
- An already implemented rule execution engine, which takes care of the matching of rules, or a concise special-purpose language for this task.
- Declarative pattern matching and rewriting, instead of nested loops iterating incident edges or adjacent nodes.
- A graphical debugger, which allows the developer to stepwise follow the rule executions, highlighting the currently matched rule in the graph.
- A graphical rule editor.
- An already implemented and tested library for predefined high-level tasks.
- Higher performance than an unoptimized manually implemented solution.

Potential problems. There are also possible drawbacks in using a transformation tool compared to a manually implemented solution:

- The time needed to learn the tool and its languages. The higher productivity achieved when using a tool is offset by the initial effort to learn the tool.
- Reduced flexibility w.r.t. transformations implemented in general purpose programming languages in general. Especially missing flexibility regarding tasks the tool was not designed for; they might lead to complicated solutions. This holds in particular for the case that the requirements change after the project start.
- Increased deployment effort due to the transformation engine and its prerequisites.
- Potential performance problems because of an unoptimized engine. This might be the case with a manually coded solution as well, but optimizing a tool supplied by others is a more difficult task.
- Maintenance problems due to the dependency on a third-party component; esp. vendor lock-in – the developer might be forced to abandon a solution in case the tool is not maintained anymore, esp. if it is closed source.
- Lack of advanced IDE features commonly offered for general-purpose programming languages, such as refactorings.

1.5. Outline of the tool comparison approach

Choosing a promising tool is aided by four kinds of information, at an increasing level of detail.

The world of transformation tools. This section introduces the prospective user into the world of transformation tools. A coarse grain map and an introductory classification of the field of transformation tools is given, explaining the major high-level discrimination points. A first reduction of the set of candidates can be carried out alongside the different approaches based on some feature matrices.

Hello World. After the introduction into the world of transformation tools, the Hello World case [55], which was posed at the Transformation Tool Contest 2011 [78] is described. Consisting of several simple yet highly illustrative tasks, it is especially well-suited for a tool comparison. The complexities of full challenges, e.g., the other ones posed at the workshop, would only hide the basics. After the case description, all tools are introduced according to the following scheme: first the tool is described in a nutshell, then the solution to an especially explanatory yet small subtask of Hello World is explained, and finally the tool vendors discuss what their tool is especially well suited for. This section extends the introductory classification with a first step into greater depth.

Transformation Tool Contest. In the following, the tools and especially the solutions are evaluated further along the votes cast for the solutions by the tool vendors at the Transformation Tool Contest (TTC). For this, first the TTC is introduced and the role of SHARE [20] explained, then the votes are listed, and finally discussed and put into perspective.

Classification in detail. The next section gives an in-depth-classification regarding a multitude of questions a user faces when deciding in between tools, they are answered with feature matrices. Before each matrix a motivating questions is asked, then the features are introduced, and finally the tools are compared regarding their support for the specified features. This allows to quickly reduce the large set of tools according to the criteria of the task at hand to a small set of tools to be evaluated in depth.

After having reduced the large set of tools to a small set of ordered candidates, one can now evaluate them in depth, based on their article in the proceedings of the workshop [78], but especially by having a look at their SHARE images.

1.6. Contribution

The contributions of this article are as follows:

- It helps software engineers, which are facing a model transformation or graph rewriting problem in choosing a tool that is well-suited to their needs; for this it gives for one an overview and for the other a comparison of many of the state-of-the-art graph- and model-oriented tools based on feature matrices. The comparison includes more tools than any other comparison known to us, which was made possible by the large number of submissions to the Hello World case [55] posed at the Transformation Tool Contest 2011.

- It discusses the Hello World case, the solutions of the tools to the Hello World case, and the ratings of the solutions by the audience at the transformation tool contest and puts them into perspective. All of this should allow to pick a suitable tool much faster than reading through numerous papers describing the tools, or the detailed case and solution descriptions, which were published in the Proceedings of the Transformation Tool Contest [78].
- It describes how SHARE [20] was utilized during the Transformation Tool Contest, giving an example how research events (especially their reviews) can be improved with this software system. Furthermore, it links to the SHARE images, which were produced by the tool vendors for the Hello World case – they allow a prospective tool user to evaluate a tool in depth before drawing a final decision.

2. The world of transformation tools

In the following, we introduce a prospective tool user into the world of transformation tools. To this end, we ask typical user questions, which we answer with explanations, and with feature matrices listing all the tools. The explanations introduce core notions of the field, while the feature matrices give a direct overview of the tool landscape.

The introduction is organized along five areas:

Suitability What are the goals of the tool, what is it suited for?

Data Which data is to be transformed?

Computations What kinds of computations are available, how are they organized?

Languages and user interface How does the interface of the tool to the user look like?

Environment and execution How does the interface of the tool to the environment look like, how is it executed?

In this section we give an overview, a coarse grained map of the field; only the computations are already explored with an increased level of detail. The areas are investigated in more depth in Section 3 explaining the solutions to the Hello World case, and especially in Section 7, which is focused on answering questions that typically arise when one has to decide whether to use a tool or not, at the then-needed level of detail.

2.1. Suitability

"Is the tool suited to my task?" is the first question that comes to mind when deciding whether to use a tool or not. The answer depends on many sub-questions, which are investigated later, but the design goals of the tool vendors, and what they think their tool is suited for should give a first hint. In Table 1, the tools vendors specify with one line what their tool is suited for and with two additional lines why they think this holds, telling about the strong points of the tool. Everything is available on one page, so the tools can be easily compared against each other. A more detailed explanation of these points follows in Section 3.

Knowing about what the tool is suited for, or what the strong points are is very helpful, but needs to be complemented by the points the tool is explicitly not designed or suited for; things that remain often unspoken and may lead to a bad choice based solely on matching strong points. So we list them in Table 2; these points may be a direct consequence of the points the tool was designed for, e.g., the goal performance is in general hampering runtime flexibility.

2.2. Data

"Can I adequately model my domain?" is the most important user question concerning the data. A first answer is given by Table 3. There, the domain of the tool and the kind of the tool are distinguished. In Figure 1 the two prototypical ones are illustrated, to the left the graph rewrite tools, and to the right the model transformation tools.

Domain. The first discrimination point is the domain of the tool, with a distinction into graph-based tools and model-based tools.

graph In the realm of graphs, the data is called a graph and conforms to a type graph.

Suitability and strong points of the tool
ATL/EMFTVM: *general-purpose model transformation.* ATL is a mature language for mapping input models to output models. The EMFTVM runtime introduces composition and rewriting.
Epsilon:
Edapt: *model migration* in response to metamodel adaptation. High automation by reuse of recurring migration specifications. In-place transformation, seamless metamodel editor integration.
GReTL: *model transformation*
GrGen.NET: *general-purpose graph rewriting.* Pattern matching of high performance and expressiveness; highly programmable. Excellent debugging and documentation. Focus on compilers, computer linguistics.
GROOVE: *state space exploration, general-purpose graph rewriting.* Rapid prototyping, visual debugging, model checking. Expressive language (nested rules, transactions, control); isomorphism reduction.
Henshin: *graph transformations for EMF models* with explicit control flow. Expressive language (nested rules, support for higher-order transformations) JavaScript support, light-weight model & API, state space analysis
MDELab SDI: *graph transformations for EMF models* with explicit control flow. Expressive language, mature graphical editor, support for debugging at model level. High flexibility, easy integration with other EMF/Java applications.
metatools: *general-purpose model transformations.* Seamless integration of hand-written and generated sources, of imperative and declarative style. Full access to host language, libraries and legacy code.
MOLA: *general-purpose model transformations* with explicit control flow. Expressive language, graphical editor with graphical code completion and refactorings, built-in metamodel editor, EMF support.
QVTR-XSLT: *general-purpose model transformations.* Supporting the graphical notation of QVT Relations with a graphical editor to define transformations, and generate executable XSLT programs for them.
UML-RSDS: *general-purpose model transformation* with verification support. Declarative transformation specification using only UML/OCL. Efficient compiled transformation implementations.
Viatra2: *general-purpose multi-domain model transformations.* Model space with arbitrary metalevels, excellent programming API. Incremental pattern matching.

Table 1: Suitability and strong points of the tool

Subject	**Tasks the tool is less or not suited for**
ATL/EMFTVM	model-to-text-transformations, high-performance rewriting
Epsilon	
Edapt	model-to-model transformations, model-to-text transformations
GReTL	EMF support only via conversion
GrGen.NET	weak XMI (no EMF) support, no runtime metamodel/rule changes, no direct model-to-model transformation
GROOVE	model-to-model transformations, model-to-text transformations
Henshin	model-to-text transformations, any transformation where the order of links matters
MDELab SDI	non-deterministic rule applications, rules with non-local search
metatools	no EMF/Eclipse support, no dynamic typing, no runtime metamodel change
MOLA	model-to-text, runtime metamodel/rule changes
QVTR-XSLT	model-to-text, models without unique identifier
UML-RSDS	EMF support, model-to-text, runtime metamodel/rule changes
VIATRA2	EMF support, runtime rule changes

Table 2: Tasks the tool is not suited for

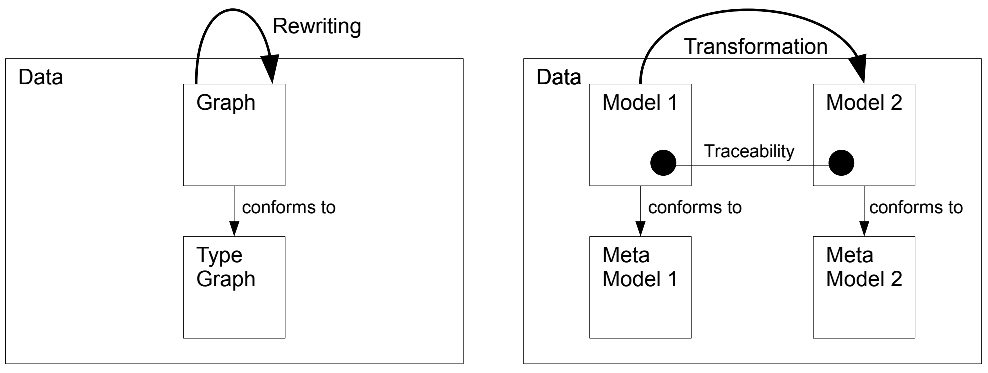


Figure 1: The data refined

model In the realm of models, the data is called a model and conforms to a metamodel.

To a certain degree the difference between graph and model transformation tools is a question of self-definition, to which community and culture the tool authors have stronger links, or what they define as their goals. Of high importance are the supported input formats and output formats defining the technical space [46] the tools operate in: model transformation tools are normally built on UML/MOF/XMI/Ecore, whereas graph transformation tools are typically built on GXL or custom formats. Graph-based tools are typically more general purpose, geared towards all domains that operate on graph like representations. In contrast, model tools are typically centered on program representations. Graph transformation tools tend to view edges as first level constructs, which might be attributed or endowed with an inheritance hierarchy, while model transformation tools tend to view edges as references between nodes without own attributes. The world of graphs, which is older and more mature, enjoys a higher level of mathematical sophistication compared to the more practically minded world of models (sometimes at the price of regarding practical usability an inferior topic).

In addition to this two opposite values, there are two refinements available:

mixed is used for tools that are located in the middle of the two domains.

multi is used for tools that offer to switch the metametamodel.

Tools normally provide one fixed metametamodel based on which the users can define the metamodel to work with. Tools classified as multi allow to switch the modeling technology to work with different metametamodels. They may show a different behavior regarding certain features we compare based on the metametamodel; in cases there the difference is of importance we hint at it with a table footnote.

The modeling technology for tools that are not using a custom model, but are built on the API of a common technology, like EMF for the Eclipse Modeling Framework [70], is given in parenthesis. The advantage of such a setup is that other tools built on that technology can be easily (re)used; a disadvantage may arise from the fact that it is not possible to adapt the model implementation to the computations, esp. that it is not possible to tweak it for more efficient matching.

Kind. The second discrimination point is the kind of the tool, with a distinction into transformation versus rewriting.

transformation A transformation tool operates on several models. It typically maps from a constant source model to a target model (and maybe further models). Transformation tools offer explicit syntactical means to distinguish between several models/instances, with built-in support for traceability information (relating source to target elements). The models might conform to the same metamodel. This way rewriting tasks can be processed by transformation tools.

rewriting A rewriting tool operates on one model. It might be a union of several metamodels, this way transformation tasks can be processed by rewriting tools.

both The tool supports both kinds.

The discrimination into rewriting and transformation gives the narrow sense interpretation of the notions – often transformation is used in a wider sense, comprising both rewriting and transformation, and in general any kind of algorithmic activity on one or multiple models; this is the notion used in the title of this article, too.

For tasks where only a small part of the model needs to be changed and the rest should stay untouched, rewriting tools are more adequate as there only the changes need to be specified and executed. In transformation tools one must specify and execute the copying of the parts, which should stay untouched. For tasks where one representation is to be mapped to another one, the transformation tools are better suited: they do not need to take care of a model partly built from elements from the source and partly from the target model, and they offer implicit traceability handling. Complicated tasks, which need to be decomposed into a series of smaller tasks tend to favor the rewriting approach: a series of rewriting steps (each responsible for a small part of the overall work) is easier to specify and more efficiently executed compared to a series of full transformations from one representation into the other. But the expressiveness of the computations we visit later on has an even stronger impact here.

Traceability. Traceability allows to fetch the source from the target element or the target element from the source element. The support for traceability links can be distinguished into:

Tool	**Domain**	**Kind**	**Traceability**
ATL/EMFTVM	model (EMF)	both	implicit
Epsilon	multi (EMF, . . .)	both	
Edapt	model (EMF)	rewriting	preserve
GReTL	graph	transformation	implicit
GrGen.NET	graph	rewriting	explicit, preserve
GROOVE	graph	rewriting	explicit
Henshin	model (EMF)	rewriting	explicit
MDELab SDI	model (EMF)	rewriting	explicit
metatools	model	both	explicit
MOLA	model	both	explicit
QVTR-XSLT	model	transformation	implicit
UML-RSDS	model	both	explicit
VIATRA2	mixed	rewriting	explicit

Table 3: Domain and Kind

implicit Traceability is built-in, source and target elements are automatically linked by the transformation engine.

explicit Traceability must be explicitly coded by assignments to variables of map type.

preserve Traceability can be modeled by keeping the identity of the transformed entity in a rewrite-based tool supporting retyping, cf. Table 19.

none None of the above.

Typically only transformation tools offer the user convenient and declarative implicit traceability. For a rewrite-based tool, it is possible to model traceability links between nodes (only nodes) with an edge in between the source and the target. This "pollutes" the metamodels but allows for a trivial visualization as a surplus (in case the tool supports visualization).

2.3. Computations

"Can I adequately specify my computations?" is the most important question concerning the computations over the data. A first answer is given in Table 4. The question should be interpreted more as a question of type "How directly can I express what I want to express?" than a question of type "Can the tool be used at all?".

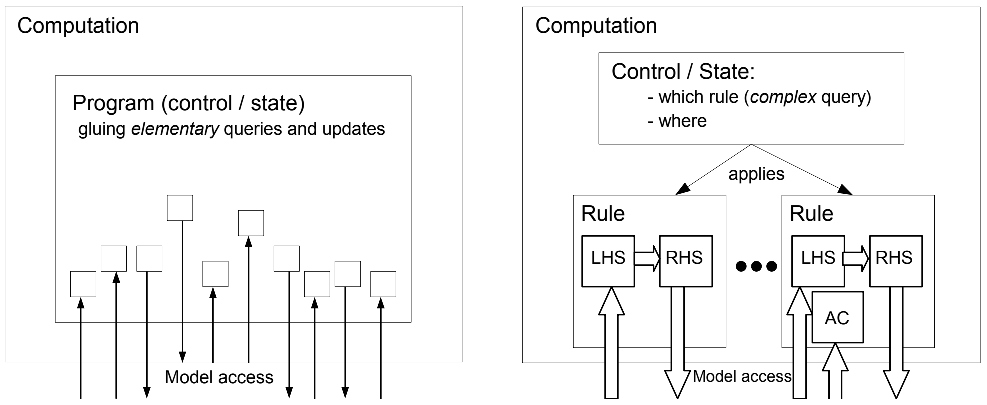


Figure 2: The computations refined

Programs. The most basic approach and the one we start our explanations with is the program-based one: the computations are implemented in an imperative object-oriented programming language, either a general-purpose programming language or a domain-specific programming language offered by a tool. The program-based approach builds on *simple queries* and *simple updates* against the API of the model, which are glued together by *state variables* and *control structures.* With simple queries we mean read operations, which return all elements of a certain node type, or the attribute values of a node, or an iterator over all incident edges/adjacent nodes of a given node. With simple updates we mean create element operations, delete element operations, and attribute value assignments. They are combined by state variables, either of basic type for storing single elements, or storages, i.e., variables of container types; and by the commonly known control structures, i.e., sequences, conditions, loops, and subprogram calls. A large transformation is built from multiple subprograms. The image to the left in Figure 2 illustrates the programmed approach.

Alternatively to a model or graph API offering operations to access all elements of a certain type, there could be variables containing root nodes as entry points; typically this leads to computations, which are organized into passes navigating the object structure following the contained or referenced elements. This is the traditional non-model way of processing object structures.

Navigational expressions. The first step away from the general-purpose programming languages and towards domain specific programming is carried out with the inclusion of OCL [88] expressions. The Object Constraint Language combines attribute comparisons with logical operations and especially with navigational expressions, which allow to formulate constraints on the connection structure. Being expressions, they are executed without side effects on the model and return a value: for references with one target element, the return value is the single element, but for references with multiple targets, a value of container type is returned (the same holds for collecting all elements of a certain type).

Rules and query-update units. A further step taken by most of the tools is the creation of declarative transformation units as central element, which consists of a *complex query* – also known as left hand side or precondition – and a *dependent update*, i.e., an update depending on the query result – also known as right hand side or postcondition. The central service offered by the transformation tools is to relieve the programmer from writing the glue code with control and state needed for implementing these complex queries and dependent updates. The tools offer a interface at a higher level of abstraction. The declarative transformation unit is distinguished into rules, in which the update is strongly coupled to the query by single element variables; exactly the elements, which were queried are available for postconditions processing. And into database query like query-update-units, which are loosely coupled: the query fills some container variables, which are then read by the update. With rules or queries-update-units, the model state is changed in big steps, one half-step reading and one half-step writing, compared to a series of small interwoven steps in the program-based approach.

Separation into layers. Another step taken by the tools building on declarative transformation units is the separation of the computations into a query plus update layer (the transformation unit) and a control plus storage layer. Only the transformation layer queries and updates the model directly. The control layer on top of it is responsible for orchestrating the rules, it defines *which* rule is to be executed next (scheduling) and *where* it is to be executed next (location). *Implicit control* is exercised by a control engine deciding which rules to apply where. One is typically able of communicating hints or constraints on rule scheduling to the execution engine, e.g., with rule priorities. We speak of *explicit control* if it is programmed by the user. The tool

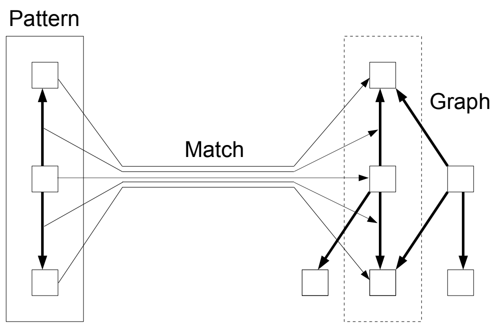


Figure 3: Pattern matching illustrated

might offer a dedicated rule control language, it might offer a tool specific programming language that is used for control, or it might be programmed in a general-purpose programming language. If rules are triggered by the engine when the model changes we speak of *event-driven* control. The image to the right in Figure 2 illustrates the rule plus control approach.

Application conditions. A further step of separation is carried out in the declarative transformation unit based tools with application conditions. They specify constraints, which must hold for applying the rule. The elements captured by them are *not* available for rewriting or mapping, though. They are typically chosen by languages mapping only one element to multiple elements – this eases to automatically maintain traceability information – to be able to constrain the location of application. In application conditions one may especially ask for elements to *not exist*. In query-and-update-based languages both parts are combined, and explicit projection of the values of interest is used instead.

Patterns. Within the rule-based approach, often *patterns* are used. They specify a subgraph, which is to be matched in the host graph, a small model which is sought after in the model, as illustrated in Figure 3. The hallmark of the pattern-based approach is the direct representation of the pattern elements inside the host graph. The semantics of patterns are based on existence, and a search for an match, binding each pattern element to a graph element; if a match is found, the rule is applied. In the pattern-based approach, rules consist of two patterns: by identifying common elements it is known what is kept, elements only available in the left hand pattern are deleted, elements only available in the right hand pattern are created.

Direct reuse. Instead of offering more complex means to combine computations and esp. queries, one could aim for direct reuse, with a library of predefined computations one chooses from and parameterizes. This approach obviously only works for constrained domains or task where some predefined routines and a few simple parameters are sufficient to define the required computations. If it can be applied, it offers the largest amount of reuse.

Helper code. Instead of stepping up the abstraction level of the computations with domain specific languages, one could stay at the level of the program-based approach and user programmed code, just aided by a code generator, which emits some helper code. The central service offered is the unfolding of a concise specification of the data and some processing aspects to boilerplate code, which would have been cumbersome to write manually, but everything else is directly programmed in a general-purpose programming language against the generated code.

Discussion of suitability. Patterns and pattern-based rules are a simple and intuitive computation approach for graph structured data, allowing for direct visualization. But this comes at a cost: the fixed structure patterns are less expressive than query or programming languages. For tasks that only require to match and rewrite fixed shapes, the solutions based on them are as concise and declarative as can be. But they fit badly to tasks that require to process all neighboring elements of a node, or that require to search for a path from a node to a target node. For these kinds of tasks, a high amount of control over tiny patterns is needed, virtually eliminating the advantages of the pattern-based approach, falling back to a programmed solution; or even worse: to a programmed solution separated into multiple layers. To counter these deficiencies, the pattern-based languages were extended with further constructs. We have a closer look at these points later on; especially as the support for processing breadth-extending structures and depth-extending structures is of high importance for all kinds of tools regarding expressiveness.

The program-based approach in contrast does not suffer from being incapable of expressing things, it is highly flexible and adaptable — it only does not offer many advantages over traditional imperative or object-oriented programming languages. The program-based approach works well and yields

Approach
ATL/EMFTVM: Rules mapping from an element and OCL expressions to elements and a connecting program with implicit control, or rules rewriting a pattern to a pattern with explicit control.
Epsilon: Task specific languages built on a base programming language (incl. OCL-like expressions). For transformations: rules mapping from an element and OCL expressions to elements and a connecting program, with implicit control.
Edapt: Library with predefined operations for adapting a model in reaction to metamodel changes, further operations are to be programmed in Java.
GReTL: Library to be extended, offering a query language yielding data containers; the updates are to be written in Java, some basic ones are predefined.
GrGen.NET: Rules rewriting patterns to patterns with explicit control.
GROOVE: Rules rewriting patterns to patterns with explicit or implicit control, and strategies for state space enumeration.
Henshin: Rules rewriting patterns to patterns with explicit control.
MDELab SDI: Rules rewriting patterns to patterns with explicit control,
metatools: Code generator for Java classes and access helpers, visitors, rewriters. The computations as such are to be programmed as visitors in Java and are carried out during visitor runs.
MOLA: Rules rewriting patterns to patterns with explicit control.
QVTR-XSLT: Rules from patterns to patterns with implicit control.
UML-RSDS: Rules which ensure a postcondition OCL expressions is satisfied when a precondition OCL expression was matched, with implicit or explicit control.
Viatra2: Rules rewriting patterns to patterns with explicit or event-driven control, or direct usage of the control language as programming language.

Table 4: Approach to specifying computations

concise solutions if the task at hand requires only small queries. This is typically the case for mostly 1:1 mapping tasks of one model to a structurally similar model, which only require to query a bit of local context. But for these kinds of tasks the transformation tools built on OCL and implicit control offer a compelling alternative. Being expressive enough for these tasks, they lead to a more declarative, functional-style specification.

If the task at hand requires to specify queries comprising more than a handful of nodes though, the query-update or pattern-based tools become a must-have in order to achieve a concise solution, as only they can evade the large amount of glue code needed in implementing the complex queries. For simple 1:1 mapping task on the other hand they may be a bit heavyweight (especially due to their typical separation into layers).

In order to decide between the pattern- and the query-based tools, which both offer declarative queries, one needs to ask "Is the update part highly context dependent?". If it is, the pattern-rule-based tools are better suited, for one can specify in finer detail what should happen in what situation with what elements, whereas the query languages must operate through the bottleneck of the query result data structures. But this might be more than offset by the expressiveness of the queries (and influenced by further factors).

A similar argument regarding simple queries for 1:1 mapping tasks and complex queries for mining of distributed data or matching large patterns holds for implicit control and explicit control: Implicit control typically allows for more concise specifications (no control program needed) for tasks which are well-suited to the control engine's strategy. Explicit control is more robust regarding problems for which the implicit control engine was not designed; it then yields faster execution times (the constraints of the task at hand can be exploited), or even allows to handle a task at all.

2.3.1. Expressiveness

We introduce the reader into the topic of expressiveness alongside an explanation tour through the approaches. Features that are referenced in the following two tables are introduced on-the-fly; if abbreviations are given in parenthesis, they are used directly in the tables. Afterwards, we compare the features and discuss their consequences. This is better understandable under the given circumstances of a strong cohesion within the approaches and the cross cutting nature of the features. For the same reason the rather detailed topic of breadth and depth structure processing is investigated here, which would fit better to Section 7.

Patterns. In addition to the main pattern (**P**), the pattern-based approach knows of negative application conditions and positive application conditions, i.e., negative patterns (**NP**), which prevent the containing pattern to match in case they can be found (a not-exists operation, from elements on which are preset from the containing pattern), or positive patterns (**PP**), which must be present. These patterns define constraints, their elements are not available for rewriting. A reduced version of them is a single negative/positive element (**NE,PE**), a generalized version is the nested negative/positive pattern (**NNP**,**NPP**); a 2nd level negative re-allows a pattern forbidden by a first level negative. The attributes are handled with additional attribute conditions (**AC**), which allow to specify logical formulas over comparisons of attribute values and constants.

Furthermore, alternative patterns may be available to allow for the matching of different substructures, or iterated patterns, which allow for the matching of a pattern as often as it is available in the graph. Typically they can be nested, then they are matched from the elements on that are preset from the containing pattern. Subpatterns allow to factor out a recurring pattern and use it from different patterns. These constructs can be used as plain application conditions comprising only a left hand side pattern. They may be endowed with an additional right hand side, then specifying a complex rule (which is applied in one step after everything was found, in contrast to a recursive call in a normal programming language that carries out matching and rewriting in each single step).

Matching structures extending into depth with patterns. Subpatterns combined with alternatives allow to recursively match structures into depth, as chains of complete patterns, but only at the price of subpattern calls with explicit parameter passing. These can be omitted in the common case that paths, i.e., only nodes connected by edges need to be matched. In this case the more concise regular and iterated path constructs may be employed. Regular path expressions allow to match an iterated path constraining the incident edges and adjacent node types on the path. They subsume the iterated path, which allows to find out whether a node is reachable from another one, without the possibility to constrain the types (a special form of this is transitive containment along containment edges). The explicit parameter passing for subpatterns is less convenient and concise than the implicit passing in regular paths, but it allows in combination with the iterated patterns to declaratively match tree structures.

Matching of breadth splitting structures with patterns. The iterated patterns, allowing to match a pattern as often as it is available in the graph, are equally expressive as nested rules, which are following the notion of rule amalgamation from theory, where a kernel rule is extended by repeated rule parts. Reduced versions are available with multi-nodes and loop-header-nodes. A multi-node allows to match a pattern and then a node incident to the pattern multiple times. A loop-header-node allows to match the header node multiple times, and then for each instance the pattern. It can be seen as a loop from a control language notationally integrated into a pattern language; especially when it can be directly assigned, which allows to follow an iterated path.

Patterns are typically matched with existence semantics, the RHS of the rule is applied on the one match of the LHS that was found. After this application it can get executed again, but only on the changed state. Alternatively, a pattern may be matched with for-all semantics, a rule is then applied on all matches found for its LHS. This alone is only a step towards matching a breadth splitting structure, as the fixed kernel part must be matched before and handed in via parameters. The for-all and existence semantics of the top level pattern can be generalized with quantified patterns: an all-quantified nested pattern is matched multiple times, an existentially quantified nested pattern once. The ability to nest such patterns to an arbitrary (but statically fixed) depth boosts the expressiveness of pattern based tools considerably.

Example. To highlight the differences in between the approaches, and to ease understanding, which might be a bit difficult given just some abstract explanations, we implement a small example query in each of them.

We query the model for all nodes `a` of type `A`, which are connected via an edge `v` of type `V` to an opposite node `b` of type `B`, but are not at the same time connected via an edge `w` of type `W` to a node `c` of type `C`. In a pattern-based language this query is expressed similar to the following specification, typically even graphically:

```
a:A -v:V-> b:B
negative {
    a -w:W-> c:C
}
```

We want to note that it is not specified how that structure is to be matched.

Declarative Query Languages. Being built for SQL like querying of graph repositories or databases (the latter being repositories that additionally offer persistence), query languages comprise a direct match regarding the question for the means available for general data querying. They allow to query graphs for connected structures without incurring side effects, and report the found data back in form of collections of tuples, i.e., container variables of a complex type.

Geared towards the extraction of information distributed over a graph, they contain built-in means for querying into depth. GReQL, the only query language which was used for solving the Hello World case, employs regular path expressions as already introduced above. Breadth structures are collected into a result set by the implicit for all matches semantics of query execution.

Example. In the query-update-based languages the example query is expressed using a term like:

```
from v: E{V}
with not (startVertex(v) -->{W} endVertex(v))
report startVertex(v), v, endVertex(v)
```

The from clause declares elements of interest that are to be found in the graph, the with part specifies the conditions that must be fulfilled for the elements of interest, so they are reported back in the way specified by the report clause. The query selects all edges `v` of type `V`, restricts them to those for which no edge of type `W` exists between `v`'s start and end vertex, and finally returns a list of (start-vertex, `W`-edge, end-vertex) triples. The query assumes that the metamodel declares that `V` is an edge type between the vertex types `A` and `B`.

OCL and non-pattern-based rules. In the non-pattern-based rules typically only one source element is matched, and OCL expressions are used as application conditions to constrain the rule application. Or the rule is described by its effects, formulated by a precondition OCL and a postcondition OCL.

OCL combines attribute conditions and their single-value semantics with navigational expressions for querying the neighboring elements (or all elements in a model); when only one neighbor is allowed by the metamodel, a single model element is returned, but otherwise a set of all neighboring elements is returned. A further nested expression can then be formulated for all elements in the container. This way it is possible to collect structures extending into breadth. Depth structures can only be followed by nesting expressions, i.e. to a statically fixed depth.

Example. In the OCL-expression-based languages the example query is expressed like that:

```
A.allInstances->select(a |
  a.V->exists(b | b.oclIsTypeOf(B)) and not
  a.W->exists(c | c.oclIsTypeOf(C)))
```

All instances of the type of the node are selected, and for each of them it is checked whether one of the `V` or `W` references leads to a node of type `B` or `C`, the results are combined with boolean operators. Here the order in which the elements are to be visited is partially fixed, the search is carried out by nested expressions.

Programs and Control. In the program-based approach, queries over the model are programmed, with elementary model queries, which are glued by statements, i.e., control flow and state variables. Depth structures can be matched by loops with a variable storing the current node; in each iteration step the variable is advanced by one edge into depth. Or by recursive calls with the current node as input parameter. Breadth structures are typically matched by loops with a variable iterating over the neighboring elements. So everything can be expressed, but everything must be expressed in solutions based on tool-supplied or external programming languages.

These looping or calling schemes may be applied from the control language of a language based on declarative transformation units, too. Besides loops iterating single-element variables one may employ container variables to store the results of queries, esp. of a statically not known number of nodes.

Example. In the program-based languages the example query is expressed like that:

```
for a in model.getAll(A)
  for b in a.V
    if b instanceOf B
      found = true
      break
  for c in a.W
```

```
    if c instanceOf C
      foundNAC = true
      break
  if found and not foundNAC
    result.add(a)
```

Here the order in which the elements are to be visited is fixed, and the search is carried out by loop statements utilizing variable assignments.

Data collection and data collection results. After these explanations now let us turn towards the tools and tables. We start with the data collection and data collection results displayed in Table 5.

Data collection. “How do I query for data?” and “How do I check for context constraints?” are questions regarding elementary subtasks of a transformation. As explained before, patterns may be used and extended with certain kinds of application conditions and usages of further patterns in varying degrees of expressiveness, or a one element query may be combined with an OCL expression – alternatively an OCL expressions may be used stand alone – or a dedicated query language may be used, or data collection and constraint checking may be realized with a manually coded program.

The pattern-based languages offer simple and declarative descriptions for statically fixed patterns; they are too simple for complex tasks and thus extended by externally combining patterns or by internally introducing more powerful elements into the patterns. This is in contrast to the OCL expressions and query languages, which offer more expressive basic elements, but always have to specify using them, even when simple patterns would be sufficient. The program-based approach offers the highest flexibility in adapting to the problem at hand, which allows for concise solutions or solutions at all for tasks, which are outside the expressiveness of the other approaches, but clearly less concise solutions for tasks, which fall inside their expressiveness; and it leads to always imperative solutions. Besides these approaches, a program may be built on task-specific pre-coded routines with pre-coded data collection in case the tasks are restricted enough for complete automation.

Regarding suitability the main question is “Is the tool expressive enough to tackle my task at hand with?”. There is a trade off between generality and simplicity involved: more general constructs allow to express more subtasks in a succinct way, but are more difficult to learn and understand, may be more difficult to visualize, and could burden simpler tasks with their unnecessary generality. When it is difficult to exactly estimate beforehand what is needed, languages of higher expressiveness are the safer bet (esp. for the pattern-based languages). This holds in addition to the already passed-by discussion of the approaches, and is intended as a forerunner to the following discussion of the suitability for breadth and depth matching.

Besides the abbreviations for the pattern based constructs already introduced at the beginning of this section, we employ in Table 5 the abbreviations **PL** to denote data collection in a programming language, **E** for the single element of a language of mapping rules, and **OCL** to denote OCL expressions.

Data collection results. "Can I store results from complicated queries for later processing?". This is a basic feature of query languages, and commonly supported by programming languages. OCL expressions are explicitly designed to be free of side effects and do not allow to store results. For the rule- and control-based languages this is a discrimination point of the control language. Storing results allows to omit repeating searches for elements that were already found, thus increasing performance. It furthermore allows to decompose a task into phases coupled by storages, which is beneficial for complex tasks. The price of this feature is a reduced declarativeness for transformations and a susceptibility to ordering effects. It comes in several degrees of strength: **none** if it is not possible to store results, single valued **variable**s, **container** variables, and **compound container**s (capable of storing records, similar to the table-valued result of a SQL query).

Depth and breadth structure handling. Structures that extend into depth by a *statically* known number of k steps can be directly described with patterns of k elements, or by OCL expressions with one nesting level per step. The same holds for structures that split into breadth with k branches. In the following, and especially in Table 6, we compare the languages of the tools in their ability to capture structures with a statically *not* known number n of depth-extending or breadth-splitting steps. In particular, we compare the languages of the tools regarding their means to denote a structure going into depth by n steps or splitting into breadth to n branches with *less than* n notational elements.

Depth. Does the task at hand require to match an iterated path, i.e., a chain of nodes linked by edges, to a statically not known depth, maybe applying some changes at its end or returning the end back? This is a common subtask in transformation, and if the answer to the previous question is yes, the

Tool	**Data collection**	**Data collection results**
ATL/EMFTVM	P, NP, PP, AC or	container
	E, OCL	none
Epsilon OL	PL, OCL	container
Epsilon TL	E, OCL	none
Edapt	library operations are applied	
GReTL	Query Language: GReQL	compound container
GrGen.NET	P, NNP, NPP, AC	container
GROOVE	P, NNP, NPP, AC	none
Henshin	P, NNP, NPP, AC	variable
MDELab SDI	P, NE, PE, PP, AC, OCL	container
metatools	PL	container
MOLA	P, NE, PP, AC	variable
QVTR-XSLT	P, PP, AC	none
UML-RSDS	OCL	none
VIATRA2	P, NNP, NPP, AC	container

Table 5: Data collection

question "How do I capture structures that are extending into depth?" needs to be taken into account when choosing a tool.

The means available for solving this subtask were already introduced with the approaches. Best suited to query for structures into depth are regular path expressions, followed by the more powerful but less concise recursive patterns, followed by the less powerful iterated path expressions. OCL expressions are not able to describe depth drilling. On the non-declarative side, we can capture depth-extending structures with loops or recursive calls. This holds for the program-based approach as well as for the control layer of a rule-and-control based language. In case structures extending an unlimited time into depth *and* into breadth, e.g. trees, are to be matched, only a combination of iterated and recursive patterns is up to the task, or a manually coded recursive subprogram. We add the suffix **RHS** to the listed solutions if we can modify the items matched into depth, e.g., reversing all edges visited.

Breadth. Does the task at hand require to match a kernel structure, plus all of certain parts of interest that are attached to it, in a statically not known number, maybe applying some changes to the branches? This is a common subtask in transformation, and if the answer to the previous question is yes, the question "How do I capture structures that are extending into breadth?" needs to be taken into account when choosing a tool.

The means available for solving this subtask were already introduced with the approaches. OCL expressions offer a concise solution with their set based semantics for capturing attached nodes, and allow to capture more complex neighboring structures with expression nesting, albeit much less concisely then. GReQL offers a concise solution with its query result sets, and allows to capture and return more complex neighboring structures. Multinodes allow to capture attached nodes concisely, but cannot be generalized to more complex attached structures at all. Loop header nodes allow to capture attached nodes concisely, but can only be generalized to more complex structures with the help of rule control. The nested rules, or iterated patterns, or quantified patterns are a bit less concise for single attached nodes, but are especially well suited to capture attached patterns.

On the non-declarative side, we can capture breadth-extending structures with loops or recursive calls. This holds for the program-based approach as well as for the control layer of a rule-and-control based language. The availability of rules which can be applied with for all semantics is a help that case. We add the suffix **RHS** to the listed solutions if we can modify the items matched into breadth, e.g., linking them to another node. OCL expressions alone are not capable of achieving this, but rules of OCL expressions can.

The abbreviations employed in Table 6 are: **Imper** for an imperative solutions with loops or calls, implemented in a programming language or a rule control language, as described before. **Reg** for regular (paths), **Rec** for recursive (patterns), **Pat** for a pattern, and **F-All** for rules applied with for-all semantics.

2.4. Languages and User Interface

"Does the user interface of the tool fit to my needs or preferences?" is the primary question concerning tool-user interaction, this includes especially the languages offered by the tool. We answer it with Table 7.

Languages. Most languages employed in solving the Hello Worlds task are separated vertically into several sublanguages (normally all the parts are used in implementing a transformation); we distinguish:

DDL A data definition language to specify the metamodel (see Subsection 2.2).

PL A programming language (as explained in Subsection 2.3).

Tool	**Depth**	**Breadth**
ATL/EMFTVM	Imper	OCL, Multi-nodes/RHS, Imper/RHS
Epsilon	Imper/RHS	OCL, Imper/RHS
GReTL	Reg-Path	Result-Set/RHS
GrGen.NET	Rec-Pat/RHS, Imper/RHS	Iterated-Pat/RHS, F-All, Imper/RHS
GROOVE	Reg-Path	Quantified-Pat/RHS
Henshin	Imper/RHS	Nested rules/RHS, Imper/RHS
MDELab SDI	Imper/RHS	OCL, F-All, Imper/RHS
metatools	Imper/RHS	Imper/RHS
MOLA	Loop-with-Header/RHS	Loop-with-header/RHS
QVTR-XSLT	Rec-Pat	Iterated-Pat
UML-RSDS	Imper	OCL/RHS, Imper/RHS?
VIATRA2	Rec-Pat	F-All, Imper/RHS

Table 6: Depth and breadth support

QU A query language (as explained in Subsection 2.3).

RL A rule language (as explained in Subsection 2.3).

CL A control language (as explained in Subsection 2.3).

Besides this, we may use **SPs** for tools that are horizontally split into several special-purpose languages, an own domain specific language for each of several goals. Only a part of their functionality falls directly inside the focused topic of this article, the transformation of structures. Furthermore, **sp** is noted down for tools that offer a user interface that can be seen as a domain specific language, but one that does not allow for general transformation programming.

Attributes. In addition to the languages concerned with structural transformation, the sub-language offered for attribute computations is given.

External languages. The question "What languages does the tool offer?" needs to be complemented by the question "What languages does the tool require?" for the parts programmed in an external programming language. Here we have to distinguish whether the computations *can* be programmed via an API in an external programming language or whether they *must* be programmed in an external language. For tools whose computations can be used via an API from the outside, we have a further look in Subsection 7.3

on the other direction of usage, i.e., for using entities from the outside inside the tool languages.

We distinguish:

Main The computations are meant to programmed in a general-purpose programming language.

Extensions The computations that are exceeding the functionality of a supplied library are to be programmed in a general-purpose programming language.

API A user program written in a general-purpose programming language can manipulate the model or call the computations via an API.

Events A user program may hook into events which are fired when the model is manipulated or a rule is matched.

For all points the programming language that needs to be employed is of interest. For the tools of the rule and control kind, we specify as a refinement whether single rules can be applied from the external program (RL) thus defining an own control layer, and/or whether control sequences or complete transformations can be applied from the external program (CL).

Employing a general-purpose programming language allows to reuse the knowledge and skills one has already acquired in programming in it – and the tools which are available for it. Everything happening is fully transparent at the level of the programming language and its debugger. But only at this level – one misses the typical advantages of conciseness and declarativeness displayed by the domain specific languages of the transformation tools; and the alternative visual style of programming offered by the pattern-based tools.

If an application is to be implemented that is built on a transformation core but has to offer in addition network interaction or a sophisticated GUI, one must employ a general-purpose programming language including libraries tuned for the latter tasks. The domain specific languages introduced in this article are not able to offer these services at all. One can still benefit from the advantages of the transformation languages if their tool offers an API that allows to use the specified transformations from the outside.

The most fundamental and beneficial form of reuse are complete transformations. A step down in the layering of Figure 2 one could use the declarative transformation units (complex queries and dependent updates) offered

Tool	**Languages offered**	**Attributes**	**External Languages**
ATL/EMFTVM	RL	OCL	API : Java
Epsilon	PL and SPs, one: RL	OCL	API : Java
Edapt	sp for migration		Extensions : Java
GReTL	QU		Main : Java
GrGen.NET	DDL, RL, CL(/PL)	Java-like	API(RL,CL), Events : C#
GROOVE	DDL, RL, CL		
Henshin	RL, CL	JavaScript	API, Events: Java/EMF
MDELab SDI	RL, CL	OCL	API, Events: Java/EMF
metatools	SPs: DDLs, RL	Java	Main: Java
MOLA	DDL, RL, CL		API : Java, C++
QVTR-XSLT	RL	XSLT	API : XSLT
UML-RSDS	RL, CL	OCL	
VIATRA2	DDL, RL, PL(/CL)		

Table 7: Offered languages and extensions

by some of languages but write the control code in a general-purpose programming language [1]. This is helpful if typical restrictions of the control languages, e.g., regarding their ability to store query results in variables, would render the task at hand overly complicated. A further step down in the layering of Figure 2 would be to write the computations to a large degree in a general-purpose language reusing only a data API plus some helping code. For some tasks this is a useful approach, but even more so is a hybrid solution, where one employs declarative transformation units for the subtasks where they fit well, and manually codes the solutions to the subtasks where they do not. Events fired on model changes finally allow for an even tighter integration of tool supplied data and computations into an externally written program. ‘

Form of Specification Language. In the following Table 8, we list the form of the specification languages. We note down:

G for graphical languages like UML class or activity diagrams.

[1] Such an architecture would be very similar to the layering applied in a lot of applications written in a general-purpose programming language, which delegate the subtask of storing data persistently and retrieving it again via an API to a database engine offering declarative SQL queries and updates. Here the programming language is notably used to glue the single SQL statements into larger activities.

T for textual languages.

Graphical languages are normally more intuitive, often better readable, and can be learned quicker. This holds especially for pattern-based languages, which offer the most intuitive encoding of structural changes – this depends on the number of elements though, if they become large the advantage of immediate visual understandability of structures fades. Textual languages are typically more concise and expressive, and can be edited more easily. They offer a better integration into existing source code management systems and their textual difference engines, but especially they allow to generate specifications by some text-emitting scripting code. For this reason even pattern-based tools – for which graphical languages are the more natural encoding – may still offer textual languages. To a certain degree the choice is a matter of personal taste, one can find out about one's own preferences by inspecting the listings in the tool descriptions in Section 4, and especially the SHARE images.

The form is applied to the:

Metamodel for the metamodel specification.

Computations for the computation specification.

The computation specification comprises one value for tools of non-rule kind, and two values for tools of the rule and control kind, first the rule, then the rule control part.

Graphical editors. In addition, we list the availability of graphical editors, differentiated in between the different language parts:

MM stands for a graphical metamodel editor, which allows to edit the metamodel in a notation similar to UML class diagrams.

RE stands for a graphical rule editor, which allows to edit the left hand side and right hand side patterns of the rules in a notation similar to UML object diagrams.

CE stands for a graphical rule control editor, which allows to edit the control flow in a notation similar to UML activity diagrams.

Tool	**M.-M.**	**Computations**	MM	RE	CE
ATL/EMFTVM	G or T	T, T	×	×	×
Epsilon	G or T	T	×	×	×
Edapt	G	G (T for extensions)	×	×	×
GReTL	G or T	T	×	×	×
GrGen.NET	T	T, T	×	×	×
GROOVE	G	G, T	✓	✓	×
Henshin	G	G, G	×[1]	✓	✓
MDELab SDI	G or T	G, G	×[1]	✓	✓
metatools	T	T	×	×	×
MOLA	G	G, G	✓	✓	✓
QVTR-XSLT	G	G, T	✓	✓	×
UML-RSDS	G	T, T	✓	✓	×
VIATRA2	T	T, T	×	×	×

[1] third-party editors for EMF-based metamodels are T or G

Table 8: Form of specification language

Graphical editors bring graphical languages to life; while a graphical language without an own editor would be unusable, a textual language, esp. a pattern-based one might come with an additional graphical editor. Graphical editors offer the benefits of visual programming, but bind the user to that editor (in addition to the underlying execution engine, as is the case for the textual languages).

2.5. Environment and execution

"In which environment can I use the tool?" and "How are the tool specifications executed?" are the primary questions we have a look at in this section.

Execution host. "How are the transformations executed which I specified?" is answered in Table 9, by the following values:

EXE The tool suite contains an external executable, which allows to run the transformation, e.g., a shell application or a simulator.

IDE The tool suite contains a plugin for an IDE that is able to execute and debug the code.

APP The transformations are executed from an user application, which is accessing the transformation via an API. (On what kind of machine?)

If the transformation should be integrated as the algorithmic core into a user application, an API is required. If the transformation service one needs consists only of the mapping of one file to another file, an external executable is the most advantageous execution host, as neither further code files are needed, nor have the startup costs of an IDE to be payed. An IDE integration offers the most convenient development, especially if a transformation consists of an external code part and a tool own part which are both supported.

Operating System. "Which operating systems are supported?" is answered by:

Win Windows,

Lin Linux or Unix,

Mac Mac OS X.

The tools supporting all three operating systems are built on the Java virtual machine or the Common Language Runtime; they can be used on all operating systems for which those platforms are available or will be available. Native programs are bound to their underlying OS unless they are ported with high effort to another one, but they typically offer performance advantages over VM-based tools, better integration into their host system, and do not require that a VM is available; this point is investigated into depth in Subsection 7.4.

Tool execution. "Is the tool able to handle my workload?" is a question whose answer depends on the workload, but a broad hint at the performance characteristics is possible by the execution model, the matching engine, and the memory consumption.

Execution model. In the following Table 10, we compare the execution model of the specification languages. We note down:

C for compilation or code generation.

I for interpretation.

Tool	**Execution host**	**Operating System**
ATL/EMFTVM	IDE(Eclipse), APP(JVM)	Win, Lin, Mac
Epsilon	IDE(Eclipse), EXE, APP(JVM)	Win, Lin, Mac
Edapt	EXE, IDE(Eclipse), APP	Win, Lin, Mac
GReTL	EXE, APP(JVM)	Win, Lin, Mac
GrGen.NET	EXE, APP(.NET)	Win, Lin, Mac
GROOVE	EXE, APP(JVM)	Win, Lin, Mac
Henshin	APP(JVM), IDE(Eclipse)	Win, Lin, Mac
MDELab SDI	APP(JVM), IDE(Eclipse)	Win, Lin, Mac
metatools	APP(JVM), EXE	Win, Lin, Mac
MOLA	IDE(Eclipse), EXE, APP(JVM, x86)	Win
QVTR-XSLT	EXE, XSLT	Win, Lin, Mac
UML-RSDS	API(JVM)	Win, Lin
VIATRA2	IDE(Eclipse), EXE, APP(JVM)	Win, Lin, Mac

Table 9: Execution host and supported OS

Compilation results in general in higher execution speed, at the price of having the parts implemented this way being fixed at compile time. Interpretation in contrast allows for faster development turn-around times and gives the flexibility of runtime adaptation. While compilation yields faster results for frequently executed operations, it might fall back behind interpretation if queries are only to be executed once.

The execution model is applied to the:

Metamodel for the metamodel specification.

Computations for the computation specification.

The computation specification comprises one value for tools of non-rule kind, and two values for tools of the rule and control kind, first the rule, then the rule control part.

Engine. Querying for graph information, esp. matching patterns (also known as subgraph isomorphy solving) is the most expensive operation in transformation tools, thus it requires special attention; there are different approaches with different performance characteristics existing:

SB for search-based – queries are answered or patterns are matched by a search in the graph-based on a plan scheduled by the search engine. A discrimination point within this approach is the time when the search plan is computed, a static one is computed at specification time, a dynamic one can be (re-)computed at runtime to better fit to a currently given model.

INC for incremental – all matches of all rules are stored in rete-network, model changes percolate through this network to yield all matches of all rules after the model change. This allows for very fast queries at the price of slow updates and increased memory consumption. The less rules, the better the performance.

UP for user programmed – patterns are sought by nested navigational expressions or nested loops; here the user can easily optimize the matching order, but on the other hand, the user is always forced to define the matching order.

Memory. Besides the time needed to execute the transformation, the memory consumed by the metamodel is of importance. So we give the amount the memory required by a plain node without attributes, followed by the amount of memory required by a plain edge without attributes, followed by the additional amount of memory required by an 4-byte integer attribute of a node. This allows an user to estimate the memory consumption of the task at hand.

3. Hello World Case

The "Hello World!" case consists of a set of primitive tasks, each of which can be solved with just a few lines of code with most transformation tools.[2] It has resulted in an extensive set of small transformation programs that are very instructive for beginners, which is in full accordance to the goal of the TTC of comparing transformation languages and tools, and very helpful for this article centered on tool comparison regarding basic properties. The aim of the case has been to cover the most important kinds of primitive operations on models, i.e., create, read, update and delete (CRUD). To this end, tasks such as a constant transformation, a model-to-text transformation,

[2]Actually, the case has been inspired by the popular "Hello World!" programs in books about programming languages: http://en.wikipedia.org/wiki/Hello_world_program

Tool	M.-M.	Comp.	Engine	Memory
ATL/EMFTVM	C	C, C	SB(stat), UP	48/32bytes
Epsilon	C or I[1]	I	UP	not tool defined[1]
Edapt	I	I and C	UP	48bytes, 40bytes
GReTL	C or I	I	SB(dyn)	36bytes, 60bytes
GrGen.NET	C	C, C or I	SB(dyn)	44bytes, 56bytes, 4bytes
GROOVE	I	I, I	SB(stat) or INC	40bytes, 48bytes, 48bytes
Henshin	I	I, I	SB(dyn), UP	not tool defined[2]
MDELab SDI	C or I	I, I	SB(dyn)	not tool defined[2]
metatools	C	C		
MOLA	C	C, C	SB(stat)	not tool defined[2]
QVTR-XSLT	C	C, C	SB(dyn), UP	
UML-RSDS	C	C, C		
VIATRA2	I	I, I	SB or INC	1kbyte, 1kbyte, 3kbytes[3]

[1] depends on underlying modeling technology
[2] depends on EMF implementation
[3] due to canonical model representation

Table 10: Execution model

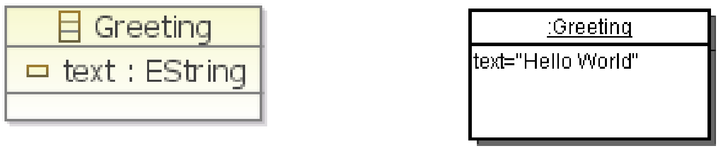


Figure 4: The "Hello World" metamodel and the example instance

a very basic migration transformation or diverse simple queries or in-place operations on graphs had to be solved.

In the following sections, the tasks comprised by this case are introduced. Note that certain subtasks had been marked as optional, i.e., those have not been required to solve the case but can be considered only as extensions. The very first task is copied verbatim from the task description. The explanations of the following tasks are only digests of the original descriptions found in [55].

3.1. Constant transformation and model-to-text transformation

1(a) Provide a *constant transformation* that creates the example instance of the "Hello World" metamodel given in Figure 4.

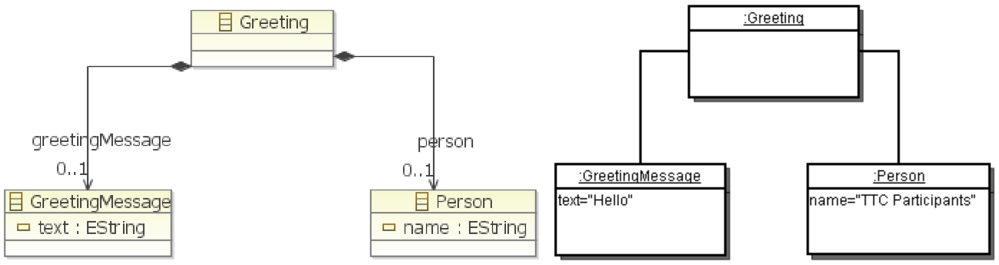


Figure 5: The extended “Hello World” metamodel and the example instance

1(b) Consider now the slightly extended metamodel given in Figure 5. Provide a *constant transformation* that creates the *model with references* as it is also shown in Figure 5.

1(c) Next, provide a *model-to-text transformation* that outputs the GreetingMessage of a Greeting together with the name of the Person to be greeted. For instance, the model given in Figure 5 should be transformed into the String “`Hello TTC Participants!`”.[3]

3.2. Pattern matching

The *querying* tasks asked for model queries counting certain elements in graphs conforming to the metamodel given in Figure 6. As results numbers were to be returned, wrapped into an object of a result type.

Model queries were asked for that count the number of nodes, the number of looping edges, the number of isolated nodes, the number of matches of a circle consisting of three nodes in a graph; and optionally the number of dangling edges.

3.3. Simple replacement

The *update* task required to provide a transformation reversing all edges in a graph (conforming to Figure 6).

[3]Note that we provide as accompanying material on the case website a metamodel, `Result.ecore`, that contains classes for returning primitive results such as strings or numbers.

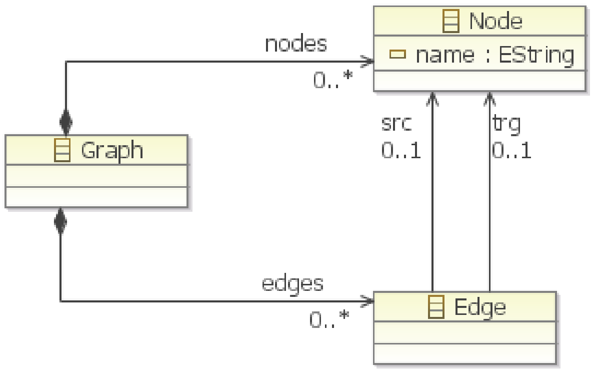


Figure 6: The simple graph metamodel

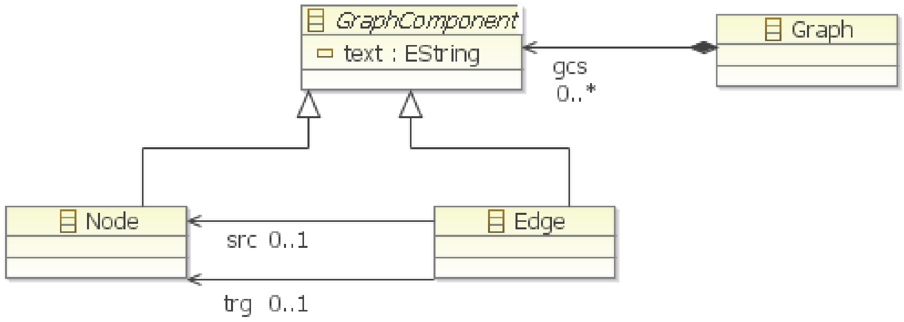


Figure 7: The evolved graph metamodel

3.4. Simple migration

The *migration* task asked for a transformation migrating a graph conforming to the metamodel given in Figure 6 to a graph conforming to the metamodel given in Figure 7. (The `name` of a node becomes its `text`. The `text` of a migrated edge has to be set to the empty string.)

An optional task was to provide a topology-changing migration that transforms to graphs as defined by the metamodel in Figure 8.

3.5. Deletion of model components

The *deletion* task required to delete a node with a given name from a graph conforming to the metamodel of Figure 6. Optionally, all its incident edges were to be deleted.

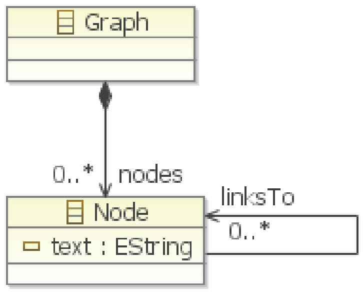


Figure 8: The even more evolved graph metamodel

Sub-task	**Control flow**	**Transformation**	**Language**	**Abstraction**	**CRUD**
Constant transformation	*output-driven*	*mapping*	*endogenous*	*horizontal*	*C*
Pattern matching	*input-driven*	*mapping*	*exogenous*	*vertical*	*R*
Simple replacement	*input-driven*	*mapping*	*endogenous*	*horizontal*	*RU*
Simple migration	*input-driven*	*mapping*	*exogenous*	*horizontal*	*CR*
Deletion	*input-driven*	*mapping*	*endogenous*	*horizontal*	*RD*

Table 11: Categorization of "Hello World" case sub-tasks.

3.6. Transitive edges

The optional transitive edges task asked for the insertion of an edge `e3` in between the nodes `n1` and `n3`, in case there was a node `n2` existing, with edges `e1` and `e2` in between `n1` and `n2` and `n2` and `n3`.

3.7. Discussion of Hello World

To provide an insight in the coverage of the "Hello World" case, Table 11 lists a number of transformation-related properties covered by each sub-task of the case. These properties are based on experiences from the TTC workshop series, as well as transformation tool surveys by [57] and [10], as typical distinguishing factors between the different transformation languages/tools. The properties are *control flow*, *transformation*, *language*, *abstraction*, and *CRUD*. *Control flow* refers to the kind of navigation required to generate the desired output. It can be either *input-driven*, where the transformation output can be fully quantified by the input (i.e., n output elements for each input element), or it is *output-driven*, where a specific output template structure is required, regardless of the kind and amount of input elements. *Transformation* refers to the kind of transformation that is required, and can be either *mapping* or *rewriting*. *Mapping* problems allow the transformation to be described as a relatively straightforward relationship between input and output elements. *Rewriting* problems are more complex, and require intermediate model states that are transformed in a stepwise fashion until the desired output is achieved. All transformation sub-tasks were *mapping* problems, so relatively simple. It also means all tasks *can* be solved by a *mapping-style* transformation language/tool, but are sometimes more easily solved using a *rewriting-style* transformation language/tool. This becomes clear during the discussion of the case solutions in this paper. The *language* property refers to either *endogenous* transformations, or *exogenous* transformations (i.e., either within the same language, or between different languages). The *abstraction* property refers to either *horizontal* (same abstraction level) or *vertical* (different abstraction levels) transformation problems. The *CRUD* property stands for *Create/Read/Update/Delete*, and refers to the kind of model manipulations required by the transformation problem.

4. The tools and their Hello World solutions in a nutshell

In the following, the tools are introduced with a calling card. The calling card consists of three parts: first an introduction-in-a-nutshell to the tool is given, which is then followed by an example solution of one of the tasks of Hello World, and finally terminated by a discussion of what the tool is suited for, as seen by the tool's authors.

The purpose of the example is to give an impression of the tool and its languages, of their look and feel. No amount of explanatory text or feature matrix comparison can overcome the benefits of getting a first impression by seeing a small example. Ideally that example must be small, but telling a lot. The "Delete Node with Specific Name and its Incident Edges" subtask of the Hello World task introduced in Subsection 3.5 satisfies these requirements pretty well [4]: it defines a simple rewriting that involves a small node-edge-node structure splitting into breadth, which further employs an attribute condition. So it was chosen as the common example to base on the tool illustration.

[4] It fits very well to the majority of tools, but less so to pure transformation tools or tools which are based on a library of pre-programmed operations for a constrained domain; but a perfect fit for all tools is not achievable given their wide range.

4.1. Edapt

Edapt[5] is the official Eclipse tool for migrating EMF models in response to the adaptation of their metamodel. Edapt records the metamodel adaptation as a sequence of operations in a history model [29]. The operations can be enriched with instructions for model migration to form so-called coupled operations. A coupled operation performs an in-place transformation of both the metamodel and the model. Edapt provides two kinds of coupled operations—reusable and custom coupled operations [29].

Reusable coupled operations enable reuse of migration specifications across metamodels by making metamodel adaptation and model migration independent of the specific metamodel through parameters. Currently, Edapt comes with a library of over 60 available reusable coupled operations [30]. *Custom coupled operations* allow to attach a custom migration to a recorded metamodel adaptation. The custom migrations are implemented in Java based on the API provided by Edapt to navigate and modify models.

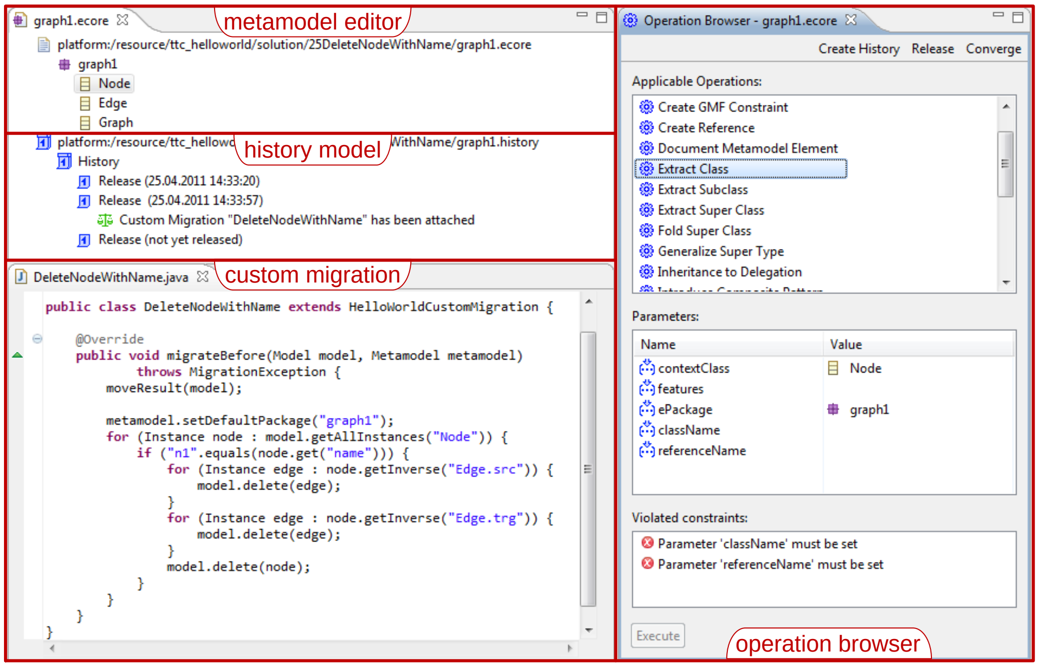
Edapt's user interface—depicted in Figure 9—is directly integrated into the existing EMF *metamodel editor*. The user interface provides access to the *history model* in which Edapt records the sequence of coupled operations. The user can adapt the metamodel by applying reusable coupled operations through the *operation browser*. When a reusable coupled operation is executed, its application is recorded in the history model. A custom coupled operation is performed by first modifying the metamodel in the editor, and then attaching a *custom migration* to the recorded metamodel changes.

4.1.1. Delete Node with Specific Name and its Incident Edges

Figure 9 shows how the history model looks like for all non-migration tasks of this case. For these tasks, the custom coupled operation always consists of a custom migration, which is attached to an empty metamodel adaptation. The custom migration is implemented as a Java class that inherits from a special super class.

Figure 9 also shows how the deletion task is implemented using the migration language provided by Edapt. The language provides methods to obtain `all Instances` of a class or `get` the value of a feature. The task can be implemented quite easily, since Edapt provides a method to `delete`

[5]`http://www.eclipse.org/edapt`



platform:/resource/ttc_helloworld/solution/25DeleteNodeWithName/graph1.ecore
- graph1
  - Node
  - Edge
  - Graph

platform:/resource/ttc_hellowo… WithName/graph1.history
- History
  - Release (25.04.2011 14:33:20)
  - Release (25.04.2011 14:33:57)
    - Custom Migration "DeleteNodeWithName" has been attached
  - Release (not yet released)

```
public class DeleteNodeWithName extends HelloWorldCustomMigration {

    @Override
    public void migrateBefore(Model model, Metamodel metamodel)
            throws MigrationException {
        moveResult(model);

        metamodel.setDefaultPackage("graph1");
        for (Instance node : model.getAllInstances("Node")) {
            if ("n1".equals(node.get("name"))) {
                for (Instance edge : node.getInverse("Edge.src")) {
                    model.delete(edge);
                }
                for (Instance edge : node.getInverse("Edge.trg")) {
                    model.delete(edge);
                }
                model.delete(node);
            }
        }
    }
}
```

Applicable Operations:
- Create GMF Constraint
- Create Reference
- Document Metamodel Element
- Extract Class
- Extract Subclass
- Extract Super Class
- Fold Super Class
- Generalize Super Type
- Inheritance to Delegation

Parameters:

Name	Value
contextClass	Node
features	
ePackage	graph1
className	
referenceName	

Violated constraints:
- Parameter 'className' must be set
- Parameter 'referenceName' must be set

Figure 9: Edapt's user interface

instances of classes. To also delete all incident edges, we can use the method `getInverse` to navigate to the edges that have the node as source or target.

Since Edapt is a migration tool, the transformation is always performed in-place. To store the result at another location, we use the helper method `moveResult` that is provided by the superclass `HelloWorldCustom-Migration`.

4.1.2. What is the tool suited for and why?

Edapt is tailored for model migration in response to metamodel adaptation. Edapt is based on the requirements that were derived from an empirical study on real-life metamodel histories [27]. The case study showed that metamodels are changed in small incremental steps. As a consequence, Edapt records the model migration together with the metamodel adaptation, in order not to lose the intention behind the metamodel adaptation.

Moreover, the study revealed that a lot of effort can be saved by reusing migration specifications across metamodels, motivating the need for reusable coupled operations. The migration tasks can be solved by applying only reusable coupled operations. Thereby, not a single line of custom migration code needs to be written. However, the study also showed that in rare cases the migration specifications can become so specific to a certain metamodel that reuse makes no sense. For these cases, Edapt provides custom coupled operations in which the migration is implemented using a Turing-complete language [28]. This language is based on Java, since we can rely on Java's abstraction mechanisms to organize the implementation, and on the strong Java tooling to implement, refactor and debug the solution.

4.2. Henshin

Henshin [1] is a high-level graph rewriting and model transformation language and tool environment for Eclipse. The transformation language of Henshin is based on declarative and procedural features. Complex transformations can be specified in a modular and reusable way using nested rules and a small set of control-flow structures. The specification of transformations is supported by a compact visual syntax provided in a graphical editor. For formal analysis, Henshin includes a state space generation and model checking tool, and an export functionality to external model checkers and other graph rewriting tools, such as AGG [71].

4.2.1. Delete Node with Specific Name and its Incident Edges

Figure 10 shows an example rule in Henshin for deleting a node with a given name and all its incident edges (modeled here by nodes of type `Edge`). The parent graph and the node name are supplied as rule parameters. To implement the deletion of all incident edges, the rule contains two nested rules, called `incoming` and `outgoing`, which are matched and applied as often as possible. Note that these two nested rules are specified indirectly with the stereotypes used on nodes and edges, e.g., ⟪`delete*/incoming`⟫. The nesting of such rules is specified using a simple path-like syntax, e.g., ⟪`delete*/x/y/z`⟫, where `x`, `y` and `z` are the names of nested rules. This allows the user to define complex transformations in a single rule and without the need for any control-flow structures, such as loops. However, control-flow structures as well as composite application conditions and attribute calculations based on scripting languages are also supported.

4.2.2. What is the tool suited for and why?

Henshin targets the transformation of structural data models in the Eclipse Modeling Framework (EMF). EMF is an implementation of a subset of the

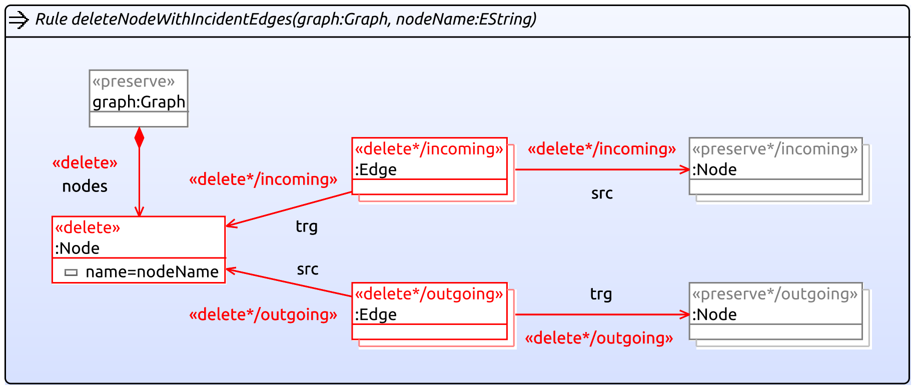


Figure 10: Henshin rule for deleting a node and its incident edges.

MOF standard by the OMG and is the basis of a wide range of tools and frameworks, including editors for domain-specific languages as well as an implementation of UML 2. Henshin is suited to define transformations for these languages. Since Henshin implements a rewrite approach, it can be used to modify models in-place, such as required for refactorings. Additionally, Henshin provides a generic trace model to support the specification of transformations from one language to another. For example, Henshin has been used in an industrial case study [36] to implement an automatic translation of programming languages for satellite technology to a standardized satellite control language, called SPELL. Due to its state space generation facilities, Henshin can be also used for formal verification. For instance, Henshin has been used for a quantitative analysis of a probabilistic broadcasting protocol for wireless sensor networks [45].

4.3. EMFTVM

The EMF Transformation Virtual Machine (EMFTVM) [87] is a runtime engine for the ATL Transformation Language (ATL) [37]. Apart from mapping a set of read-only input models to a set of write-only output models – the default execution model for ATL – it supports in-place rewrite rules. The rewrite rules are written in the textual SimpleGT language, and are compiled to the same EMFTVM byte code as ATL. Trace models are generated implicitly, and can be inspected at runtime.

For the Hello World case, solutions written in both ATL and SimpleGT are provided. While none of the transformation tasks is complex enough to warrant the use of composed ATL and SimpleGT modules, the tasks do show which are the strong points of both ATL and SimpleGT. Because both languages can be composed in a fine-grained way in the VM, one can choose which transformation rules to write in which language for each transformation sub-problem.

4.3.1. Delete Node with Specific Name and its Incident Edges

As ATL maps input models to output models, the ATL solution copies the input graph to an output graph, minus Node “n1” and its incident edges:

```
module graphDeleteN1Incident;
create OUT : Graph from IN : Graph;

helper context Graph!Edge def : targetsN1 : Boolean = self.trg.isN1;
helper context OclAny def : isN1 : Boolean = false;
helper context Graph!Node def : isN1 : Boolean = self.name = 'n1';
helper context OclAny def : notN1 : Graph!Node =
 if self.isN1 then OclUndefined else self endif;

rule Graph {
 from s : Graph!Graph
 to t : Graph!Graph (
  nodes <- s.nodes->reject(n | n.isN1),
  edges <- s.edges->reject(e | e.targetsN1))
}
rule Node {
 from s : Graph!Node (not s.isN1)
 to t : Graph!Node (
  name <- s.name)
}
rule Edge {
 from s : Graph!Edge (not s.targetsN1)
 to t : Graph!Edge (
  src <- s.src.notN1,
  trg <- s.trg.notN1)
}
```

The transformation module `graphDeleteN1Incident` creates the `OUT` model from the input `IN` model. Both models conform to the `Graph` metamodel. The paths to the actual (meta-)models are given as runtime parameters. Three helper attributes are defined to abbreviate OCL expressions, as well as to cache the expression values: `targetsN1`, `isN1`, and `notN1`. These helper attributes are evaluated in the transformation rules: `Graph`, `Node`, and `Edge`.

The `Graph` rule copies all input elements `s` that are instances of the metaclass `Graph` in the metamodel `Graph`. Each `s` is copied to an output element `t`, which is also an instance of `Graph`. Only the references to `nodes` that are not “n1”, and the references to `edges` that do not target “n1”, are copied. The Node rule copies all input elements s that are instances of the metaclass Node in the metamodel Graph, but are not “n1”. The Edge rule copies all input elements s that are instances of Edge, but do not target an “n1” node. Any edges originating from “n1” will dangle, as the src and trg references of each edge t will be set to OclUndefined as soon as the src or trg reference of edge s points to “n1”.

The SimpleGT solution for the same problem looks as follows:

```
module graphDeleteN1IncidentInplace;
metamodel Graph;
transform g : Graph;

rule DeleteIncidentEdge {
 from e : Graph!Edge (trg =~ n1),
  n1 : Graph!Node (name =~ 'n1')
 to n1 : Graph!Node (name =~ 'n1')
}
rule DeleteN1 {
 from n1 : Graph!Node (name =~ 'n1')
}
```

The transformation module graphDeleteN1IncidentInplace rewrites the model g, which conforms to the metamodel Graph. The Delete-IncidentEdge rule is applied *as-long-as-possible* to all input patterns (e, n1), where n1 is an “n1” node, and e is an edge targeting n1. The output pattern reflects the state of the input pattern after the rule is applied: e is no longer there. Then, the DeleteN1 rule is applied *as-long-as-possible* to all n1 nodes. There is no output pattern, so the entire input pattern match is deleted.

4.3.2. What is the tool suited for and why?

EMFTVM focuses on reuse, modularization, and composition of model transformations. One way to compose model transformations is to compose modules of transformation rules, and execute the composition as one transformation (internal composition). This kind of composition can provide fine-grained semantics, as it is part of the transformation language, but often applies to that transformation language only.

EMFTVM provides cross-language internal composition by defining the composition mechanism at the VM level. The VM provides a common, executable semantics for (composition of) transformation modules and rules. Two internal composition mechanisms for rule-based transformation languages are generalized in this way: module import and rule inheritance. The EMFTVM is based on the Eclipse Modeling Framework (EMF). As a result, the proposed composition mechanisms are specific to EMF. EMFTVM currently provides compilers for ATL, SimpleGT, a minimal graph transformation language on top of EMF, and EMFMigrate [86], a model migration language for EMF.

4.4. Epsilon

Epsilon is a component of the Eclipse Modeling Project[6] and a family of model management languages. Epsilon seeks to capture patterns of – and best practices for – model management. Specifically, Epsilon provides several inter-related task-specific languages. Each language makes idiomatic patterns and concepts that are important for a specific model management task. For example, Epsilon Flock [65] provides constructs for updating a model in response to changes to its metamodel.

To solve the Hello World case, three Epsilon languages were used. The Epsilon Object Language (EOL) [43] – which is the base language of Epsilon and is an extension to and reworking of OCL – was used for direct model manipulation, Epsilon Flock was used for model migration and rewriting, and the Epsilon Generation Language [67] was used for model-to-text transformation. For each problem in the Hello World case, we have chosen the Epsilon language that provides constructs that are tailored to solving that category of problem. Opponents at the TTC workshop remarked that the Epsilon solutions are concise and readable, and these characteristics of the solutions are probably due to our use of Epsilon's task-specific languages.

4.4.1. Delete Node with Specific Name and its Incident Edges

We solved the node deletion task with EOL [43] and with Flock. The latter is more concise, but arguably more difficult to understand. In EOL, the `delete` keyword removes a model element and all nested model elements from a model. To delete a `Node` and its incident `Edges`, three deletes have been used (Listing 1) because, in the graph metamodel provided by the case, a `Node` does not contain its `Edges`.

```
var n1 : Node = Node.all.selectOne(n|n.name == "n1");

delete n1.incoming();
delete n1.outgoing();
delete n1;

```

[6]http://www.eclipse.org/epsilon

```
operation Node incoming() : Collection(Edge) {
 return Edge.all.select(e|e.trg == self);
}

operation Node outgoing() : Collection(Edge) {
 return Edge.all.select(e|e.src == self);
}
```

Listing 1: Deleting a node and its incident edges with EOL.

An alternative solution, using a Flock migration strategy, is shown in Listing 2. Like all of the task-specific languages in Epsilon, Flock re-uses and extends EOL with additional language constructs. For example, Flock provides the `delete` construct for specifying model elements that should be removed for a model. Deletions are guarded using the `when` keyword. The difference between the two solutions is subtle, but important. The Flock solution is declarative, and the Flock execution engine consequently has complete freedom over how the deletion of the node and edges is scheduled. In contrast, the EOL solution is imperative, and the EOL execution engine must first find the "n1" node, then delete its edges, and then delete the node itself.

```
delete Node when: original.name == "n1"

delete Edge when: original.src.name == "n1" or original.trg.name == "n1"
```

Listing 2: Deleting a node and its incident edges with EOL.

4.4.2. What is the tool suited for and why?

Epsilon appears to be well-suited for many common model management tasks. This claim is supported by the use of Epsilon by numerous industrial partners, including in ongoing collaborations with BAE Systems [9], IBM Haifa, Telefonica, Western Geco, Siemens, and the Jet Propulsion Laboratory at NASA. Additionally, the Universities of Texas, Oslo, Kassel and Ottawa teach MDE by using Epsilon. A further benefit of Epsilon is that it is technology-agnostic: model management operations written in Epsilon languages are independent of the technology used to represent and store models. Epsilon can be used to manage XML, EMF, MDR, BibTeX, CSV, and many other types of models.

Due to its task-specific languages, Epsilon is well-suited to solving a range of model management problems, such as model transformation, merging, comparison, validation, refactoring and migration. The opponents assigned to the Epsilon solution described in this paper remarked that most of the solutions to the Hello World are very concise and readable when formulated with Epsilon. Counter to this, one of the opponents suggested that learning the similarities and differences between the family of languages might be a challenge for new users of Epsilon.

At present, Epsilon is not well-suited for matching complicated patterns. The opponents remarked that solutions that required matching complicated patterns (such as finding cycles of three nodes in a graph) were less concise and readable due to the use of imperative constructs for specifying patterns in EOL. We are currently investigating the possibility of adding pattern matching constructs to EOL. Additionally, not all of the Epsilon languages scale well for very large models in some situations. We have tailored Epsilon for use with large models to solve the specific problems of industrial partners (e.g., [9]), and we are beginning to address more general issues of scalability in Epsilon as part of our ongoing research at York (e.g., [2]).

4.5. GReTL

GReTL (*Graph Repository Transformation Language*, [34, 13]) is a graph-based, extensible, operational transformation language. Transformations are either specified in plain Java using the GReTL API or in a simple domain-specific language. GReTL follows the conception of incrementally constructing the target metamodel together with the target graph, a feature distinguishing it from most if not all other transformation languages. When creating a new metamodel element, a set-based semantic expression is specified that describes the set of instances that have to be created in the target graph. This expression is defined as a GReQL query [12] on the source graph.

For transformations with pre-existing target metamodel like in the Hello World case, there are also operations with the same semantics working only on the instance level.

GReTL is a kernel language consisting of a minimal set of operations, but it is designed for being extensible. Custom higher-level operations can be built on top of the kernel operations. This extensibility was exploited to add some more graph-replacement-like in-place operations for solving the Compiler Optimization case [8].

4.5.1. Delete Node with Specific Name and its Incident Edges

As said, GReTL can be extended. To compete in the TTC Compiler Optimization case, several in-place-operations with semantics similar to graph replacement systems.

The following operation call deletes all nodes of type `Node` whose `name` attribute equals “n1” and its incident edges.

```
transformation DeleteNodeN1AndIncidentEdges;

Delete <== from n: V{Node}
           with n.name = "n1"
           reportSet n, n <--{src, trg} end;
```

The first line is simply the declaration of the transformation. In line 3, the `Delete` operation is invoked. It receives a (possibly nested) collection of elements to be deleted. Those are specified with the GReQL query following the arrow symbol.

In TGraphs, deleting a vertex also deletes incident edges; there cannot be dangling edges. However, since the example model represents edges as vertices of type `Edge` that refer to their start and end `Node` with `src` and `trg` edges, a regular path expression is used to select the `Edge` nodes that start (`src`) or end (`trg`) at the `Node n` to be deleted as well.

4.5.2. What is the tool suited for and why?

For transformation tasks similar to the Program Understanding case [32], where complex non-local structures have to be matched in the source graph, GReTL is especially well-suited due to GReQL's regular path expressions.

4.6. GrGen.NET

GrGen.NET is an application-domain-neutral graph rewrite system with a focus on performance, programmability, expressiveness, and debugging. It offers textual languages for graph modeling and rule specification, which are compiled into .NET assemblies, and a textual language for rule application control, which is interpreted by a runtime environment. The user interacts with the system via a shell application and a graph viewer (alternatively via an API) allowing for graphical and step-wise debugging.

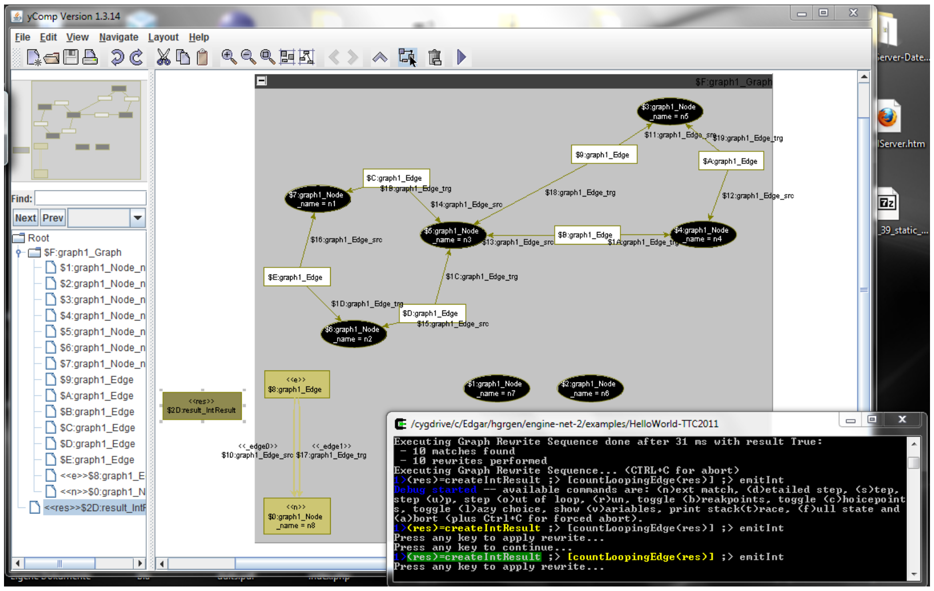
4.6.1. Delete Node with Specific Name and its Incident Edges

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The example rule `deleteN1AndAllIncidentEdges` below matches a node `n` of type `graph1_Node`, `if` it bears the name searched for. The incident edges are collected with the `iterated` construct, which munches the contained pattern eagerly as long as it is available in the graph and not yet matched. The contained pattern here consists of a graphlet `n <- e:graph1_Edge` that specifies an anonymous edge of type `Edge` leading from the source node `e` to the target node `n`. The rewrite part is specified by a `replace` block nested within the rule; graph elements declared in the pattern, not referenced in the replace-part are deleted. Since the `replace` parts are empty, all matched elements are deleted.

```
rule deleteN1AndAllIncidentEdges {
    n:graph1_Node;
    if {n._name == "n1";}

    iterated {
        n <-- e:graph1_Edge;

        replace { }
    }

    replace { }
}
```

The rule is executed from the rule application control language with the syntax `[deleteN1AndAllIncidentEdges]`. The all-bracketing ensures the rule is applied on all matches in the host graph.

When the rule is executed in the debugger of the shell, one can watch how it is applied on the host graph as illustrated by the screenshot shown in Figure 11.


yComp Version 1.3.14
File Edit View Navigate Layout Help
Find:
Next Prev
Root
$F:graph1_Graph
$3:graph1_Node _name = n5
$7:graph1_Node _name = n1
$5:graph1_Node _name = n3
$4:graph1_Node _name = n4
$6:graph1_Node _name = n2
$1:graph1_Node _name = n7
$2:graph1_Node _name = n6
$0:graph1_Node _name = n8
<<res>> $2D:result_IntResult

/cygdrive/c/Edgar/hgrgen/engine-net-2/examples/HelloWorld-TTC2011

```
Executing Graph Rewrite Sequence done after 31 ms with result True:
 - 10 matches found
 - 10 rewrites performed
Executing Graph Rewrite Sequence... (CTRL+C for abort)
1>(res)=createIntResult ;> [countLoopingEdge(res)] ;> emitInt
Debug started -- available commands are: (n)ext match, (d)etailed step, (s)tep,
step (u)p, step (o)ut of loop, (r)un, toggle (b)reakpoints, toggle (c)hoicepoint
s, toggle (l)azy choice, show (v)ariables, print stack(t)race, (f)ull state and
(a)bort (plus Ctrl+C for forced abort).
1>(res)=createIntResult ;> [countLoopingEdge(res)] ;> emitInt
Press any key to apply rewrite...
Press any key to continue...
1>(res)=createIntResult ;> [countLoopingEdge(res)] ;> emitInt
Press any key to apply rewrite...
```

Figure 11: TODO: show a screenshot from delete node with incident edges

4.6.2. What is the tool suited for and why?

GrGen.NET is suited for general-purpose graph rewriting. That claim is supported by its rich metamodel, by the expressiveness of the rule language, which allows to stay declarative for structures extending into depth and into breadth, where other tools need to fall back to control, or by the support for retyping of graph elements; and by the programmability offered by the control language, which allows to program even a state space enumeration, a concept not built in as such. It is used in many fields, e.g., model transformation [], computer linguistics [], architecture [], bio-chemisty [], mechanical engineering [], and compiler construction []. TODO: add citations

GrGen.NET is suited for performance critical tasks with its optimized code generator yielding high execution speed at modest memory consumption. It is well adapted to being used by external developers due to the extensive user manual [6] it offers and the languages, which were designed to be understandable to a common software engineer, with a direct representation of graphical patterns in a textual notation. Finally, GrGen supports the developer in his work with the debugger of the GrShell, offering stepwise execution of the rules and a visualization of the current graph.

GrGen.NET is not well suited for EMF/.ecore based transformations because of the name mangling applied by the importer, and the lack of an exporter; furthermore, it cannot be used in scenarios in which rules need to be adapted programmatically at runtime.

4.7. GROOVE

groove is a general-purpose graph transformation tool offering a user-friendly user interface and a very expressive rule language. Its main distinguishing feature is automatic exploration and analysis of the complete space of reachable graphs, using any of a number of search strategies and other settings. The analysis capabilities include LTL and CTL model checking [42] and Prolog queries [17]. See [18] for a recent overview of the tool and its use in practice.

In a typical usage scenario, groove is accessed through its built-in GUI, but for the analysis of predefined grammars there is a headless, command-line version available (offering better performance). A screenshot of the GUI is shown in Figure 12.

In the way of interoperability, groove supports import from and export to a number of external formats, including EMF, as well as a limited form of textual output.

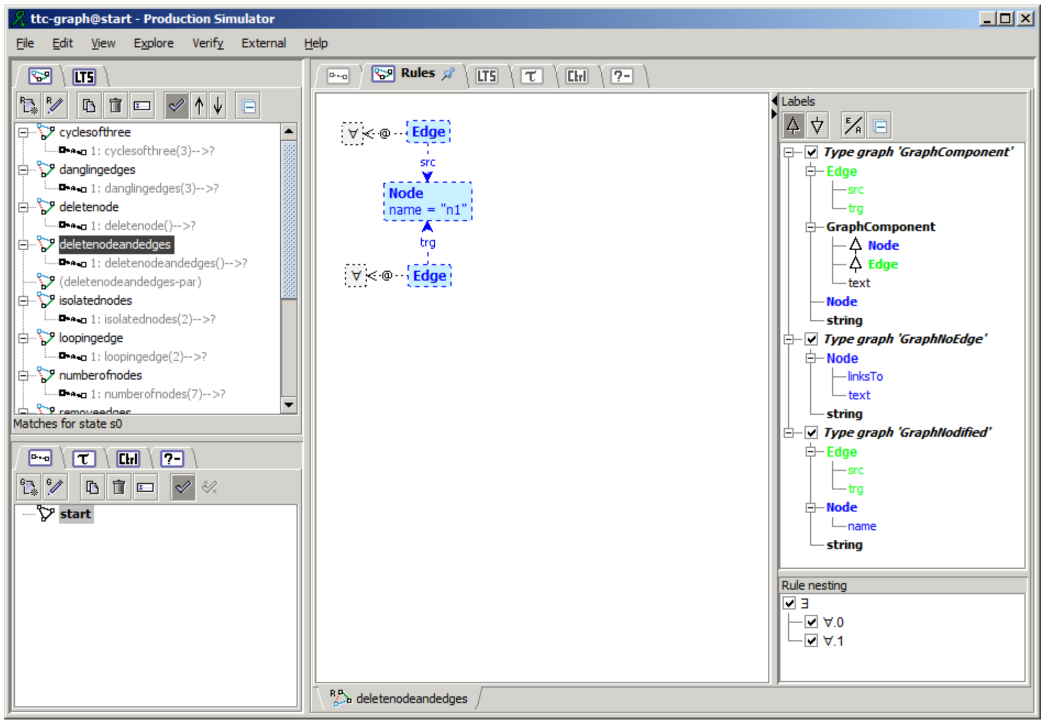


Figure 12: Screenshot of the GROOVE simulator

4.7.1. Delete Node with Specific Name and its Incident Edges

It is typical of the power of the GROOVE rule language that each challenge of the Hello World case can be solved in a single rule that captures precisely the desired functionality or effect. For instance, Figure 13 shows the GROOVE rule implementing the challenge described in Subsection 3.5, namely to delete a node with a given name (here “n1”) and all its incident edges.

The phrase “all its incident edges” gives rise to the two nodes labeled ∀ in the graph, each of which universally quantifies over the patterns connected to it by a `@`-labeled edge. Thus, the upper ∀-node captures all incoming edges, and the lower one all outgoing edges. The dashed (blue) outline of the nodes is the visual representation of the fact that they are deleted when the rule is applied.

4.7.2. What is the tool used for and why?

Experience has shown that GROOVE is very easy to use for prototyping all kinds of systems. In essence, any scenario involving the dynamics of a

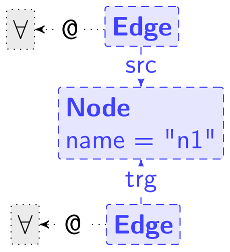


Figure 13: GROOVE rule for deleting a node and its incident edges

system that has a natural representation as graphs is amenable to modeling in GROOVE.

The advantage of building such a model is especially the ease in which the resulting behavior can be analyzed, both through simulation and visualization and through the automatic exploration of the set of reachable graphs. Central to this capability is the notion of a *labeled transition system* (LTS), which shows precisely how graphs may evolve under rule application: in this view, every reachable graph is itself a node (i.e., state) in the LTS, and every rule application corresponds to an edge (i.e., a transition).

In settings where the issue is not to model the dynamics of a system but to specify a (mode-to-model) transformation, GROOVE is still a useful tool as it can show, on concrete example models, that the rule system specifying the transformation is confluent and terminating — namely, this is the case if and only if the LTS is acyclic and has only a single terminal state.

The Hello World case offers no opportunity to demonstrate this capability, but in [18] we review examples from several, very distinct domains.

4.8. MDELab SDI

Story diagrams [15, 84] are a visual domain specific language similar to UML activity diagrams with an easily comprehensible notation for expressing graph transformations. Activity nodes and edges describe control flow and so-called story nodes can be embedded, which contain graph transformation rules.

Story diagrams can be interpreted by the MDELab Story Diagram Interpreter (SDI) [19], which features a dynamic pattern matching strategy when executing the graph transformation rule inside a story node. At runtime, this strategy adapts to the specifics of the instance model on which the graph pattern matching is executed to improve performance. Furthermore, the SDI provides seamless integration with EMF and supports OCL to express constraints and queries in a story diagram. A debugger allows to execute story diagrams step-wise, to inspect and modify the state and the diagram itself, as well as to visualize the execution.

4.8.1. Delete Node with Specific Name and its Incident Edges

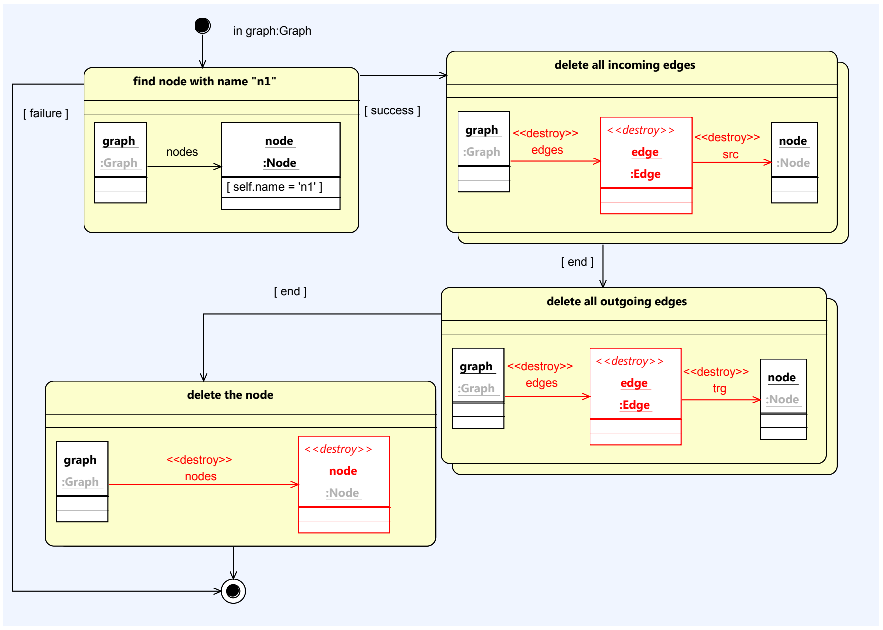


Figure 14: Searching for node *n1*, deleting its incoming and outgoing edges, and the node itself.

The problem to delete a particular node with its incident edges can be split into four steps: (1) Finding the node, (2) deleting the incoming edges, (3) deleting the outgoing edges, and (4) deleting the node itself. The story diagram in Figure 14 contains a story node for each of these steps. First, the *graph* that is modified has to be provided as a parameter. Then, the node will be sought. An OCL constraints is used to check the name of the node. If one can be found, its incoming and outgoing edges are deleted. This is done in the subsequent *for-each* nodes, i.e., the contained graph transformation rules are executed for all matches that can be found. Finally, the node itself is deleted. Instead of hard-coding the name of the node to delete, it is also possible to provide the name as an additional parameter.

Arguably, this solution is much more verbose than the solutions of other, especially textual, languages. However, the advantage of this graphical notation becomes clear if the patterns are complex and contain many links. Then, a corresponding textual representation would be much harder to understand.

4.8.2. What is the tool suited for and why?

As already stated, a major strength of story diagrams is their good comprehensibility. This is supported by allowing to use OCL expressions in any place, where constraints or queries can be used. They can be executed by an interpreter, which offers high flexibility because it allows to generate story diagrams at runtime and execute them right-away. This feature is used, e.g., in [16], where behavior models are derived from requirement specifications. The interpreter simulates different scenarios in several iterations and derives a valid story diagram, which reflects and fulfills the requirements. Besides the interpreter, a graphical editor with model validation and a graphical debugger are also provided. Another advantage is the tight integration with EMF, which allows to use other EMF-based tools with story diagrams.

4.9. metatools

metatools take the opposite approach to dedicated execution machines: The idea is to support a declarative style of programming by integrating high-level transformation devices into a general-purpose programming language (currently: Java), as seamless as possible. Some of these devices are realized by calling the API of a run-time library, others by source code that is *generated* from a compact declaration in a dedicated domain specific language with theoretically clean semantics (“`umod`”, “W3C-DTD”, etc.).

Any structure spanned by “objects” and “references”, and additionally by non-syntactic collection aggregates like “set” and “map”, can be considered a “graph”, and can be treated as such. On the conceptual level this structure is classified with graph theory (“a-cyclic”, “rooted”, etc.). This rules the generation of source code and its usage. The concrete top-level application is individually hand-coded, is based on this generated code, but is still free to use any features of the hosting programming language and its libraries.

4.9.1. Delete Node with Specific Name and its Incident Edges

The basic concept of metatools allows very different solutions for the design of the data structures for the “hello world” test case: “node”, “edge” and “label” can be realized by many different ways using Java objects. This requires explicit design decisions by the software architects.

Here, we present one variant that seems the most elegant to us. In the share demo [51] you can find alternatives. Here, we assume that the graph is rooted, i.e., can be represented by one single node, and may contain cycles, but no "dangling" edges, and that edges are directed.

```
MODEL Model =

VISITOR 0 Rewriter IS REWRITER ;

TOPLEVEL CLASS
Node
  name          string          ! C 0/0 ;
  outgoing      SET Edge        !        V 0/0 ;
Edge
  src           OPT Node        ! C 0/0  ;
  trg           OPT Node        ! C 0/1  V 0/0 ;
END MODEL
```

```
  public static Node deleteNodeAndEdges  (final Node root,
                                          final String nodeName){
    return new Rewriter(){
      public void action(final Node n){
        if (nodeName.equals(n.get_name()))
          substitute(null);
        else
          super.action(n);
      }
      public void action(final Edge e){
        if (rewrite(e.get_outgoing())==null)
          substitute_empty();
        else
          super.action(e);
      }
    }.rewrite_typed(root);
  }
```

4.9.2. What is the tool suited for and why?

The *generative* approach of the metatools determines their applicability: Once the generation of source code is done, no further dedicated tools are required, except small runtime libraries. All integration of software components requires only the genuine protocols of the hosting language, in horizontal ("packages") as well as in vertical ("inheritance") direction.

Therefore metatools are well suited for applications that have to execute in predefined and restricted or, contrarily, in arbitrary environments.

It is best suited when legacy class definitions or extensive use of (opaque) libraries are part of a project and shall be combined with more declarative techniques in an incremental way. For instance, the "paisley" pattern matching subsystem is fully compatible with arbitrarily shaped pre-defined data [75].

It is also well suited when experienced programmers want to retain immediate use of the full range of the hosting programming language and its libraries, do not want to change their style of coding completely, and stay with the tools they are used to. Features, tools and strategies can be employed selectively, according to the user's needs and preferences.

The output of metatools is source code, which can be treated together with hand-written code in a uniform way, by humans (inspecting) and by tools (generating doc, debugging, profiling, etc.). There is no "magic behind the scene", so programmers have full control over the applications' behavior, if they want to.

metatools include advanced support for XML import, export and processing, as long as the format is given as a W3C DTD.

They have been successfully employed in different industrial and academic medium-scale professional programming projects, in the fields of compiler construction for DSLs, web content management, financial systems, etc.

Currently there is *no support* for IDE integration, and for XML type definitions beside DTD. Esp. there is *no* connection to Eclipse and to EMF. But the SHARE demo on the "compiler optimization task" [51] shows how easily a new XML based file format (here: `gxl-1.0`) can be connected to metatools.

For further information see `http://bandm.eu/metatools` or the papers on selected aspects: [52, 53, 75, 76].

4.10. MOLA

MOLA [40, 59] is a graphical general-purpose transformation language developed with a focus on comprehensibility. It is based on traditional concepts among transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed.

4.10.1. Delete Node with Specific Name and its Incident Edges

MOLA is a procedural transformation language. The MOLA procedure solving this task is given in Figure 15. The structure of a MOLA procedure is in a sense similar to UML activity diagrams. The execution order of MOLA statements is determined using Control Flows (closed arrows with dashed line). The key element of the MOLA language is a rule (gray rounded rectangle), which contains a declarative pattern that specifies the instances of the classes that must be selected and how they must be linked. The first

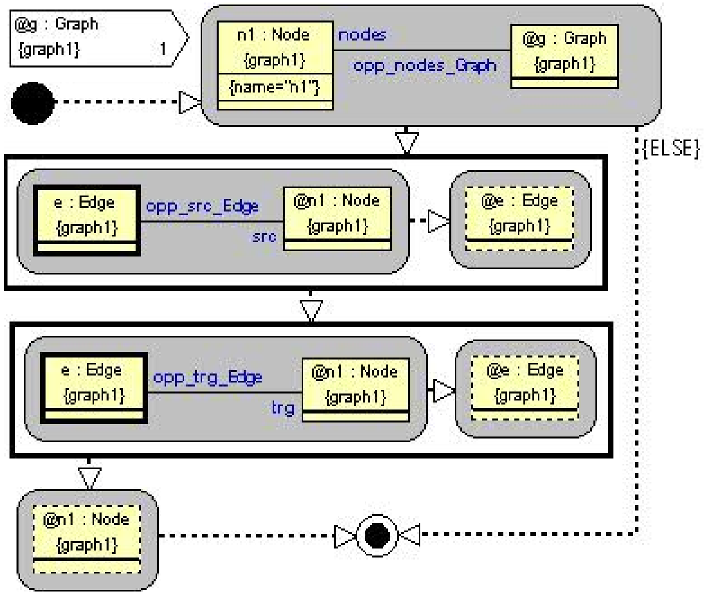


Figure 15: MOLA transformation for deleting a node named "n1" and its incident edges.

rule is used to find a `Node` named `n1` in a graph. The graph to be processed (`@g`) is given as a parameter to ensure that only nodes in this graph are examined. We check with an association link that the `Node` is contained in the `Graph`. An OCL-like constraint is used to check the `name` of the `Node`.

When the node `n1` was found, then two similar MOLA foreach loops (rectangles with bold border) are executed to process the outgoing and incoming edges respectively. As a pattern is only matched once, we have to employ a loop to process all the edges. Each loophead (first rule in a loop) contains an iterator – the loop variable, a class element depicted with a bold border. A loop is executed for each distinct instance of the loop variable satisfying constraints defined by the loophead. Here, the constraint is that the edge's source (respectively, target) is the `n1` node. The second rule in the loop deletes this edge (deletion is marked using a dashed border). Afterwards the `n1` node itself is deleted using a similar MOLA rule. Execution ends by reaching the end symbol.

4.10.2. What is the tool suited for and why?

The main design goals of the MOLA language are readability and comprehensibility of the transformations. Therefore, MOLA is a graphical transformation language. The comprehensibility of the MOLA language helps to reduce the number of errors in a transformation definition.

MOLA has rich pattern definition facilities. Therefore, the full power of the MOLA language appears if it is possible to define the solution of a task as one or few MOLA rules. This way it is possible to solve many real transformation tasks easily, where it is required to process languages with complicated metamodels like UML. This is proved by a case study performed in the ReDSeeDS project [69], where MOLA was used to process UML models in a model-driven application development. MOLA is well suited for building tools for graphical languages with complicated domain metamodels and dissimilar presentation models. An example is the MOLA editor [41] built in the METAclipse framework.

MOLA offers an Eclipse-based graphical development environment – the MOLA tool [59], incorporating all the required development support: a graphical editor (with support for graphical code completion and refactoring), a syntax checker and a compiler to three different target environments including EMF. The MOLA tool has a facility for importing existing metamodels, including the EMF (Ecore) format.

MOLA is a general-purpose transformation language, so for specialized tasks, such as model-to-text transformations or model migrations, specialized languages perform better. The same could be said about simple model navigation and look up transformations where languages such as lQuery [54] or EOL [43] perform better.

4.11. QVTR-XSLT

QVT Relations (QVT-R) is a declarative model transformation language proposed by the OMG as part of the Query/View/Transformations (QVT) standard. QVT-R has both textual and graphical notations; the latter provides a concise, intuitive and yet powerful way to specify transformations. In QVT-R, a *transformation* is defined as a set of *relations* (rules) between source and target metamodels, where a relation specifies how two object diagrams, called *domain patterns*, relate to each other. Optionally, a relation may have a pair of *when*- and *where*-clauses specified with OCL to define the pre- and post-conditions of the relation, respectively.

The QVTR-XSLT tool supports the graphical notation of QVT-R, and the execution of a subset of QVT-R by means of XSLT programs. It consists of two parts: 1) a graphical editor that is used to define the metamodels and to specify the transformations 2) a code generator that automatically generates executable XSLT programs for the transformations.

4.11.1. Delete Node with Specific Name and its Incident Edges

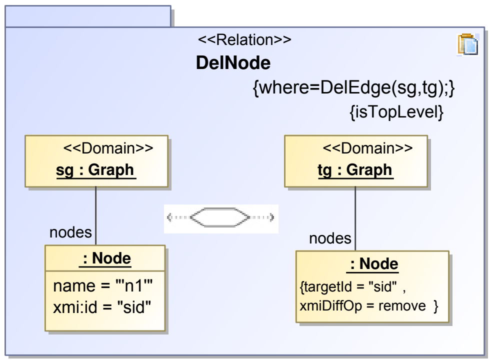


Figure 16: Delete a node with name “n1”

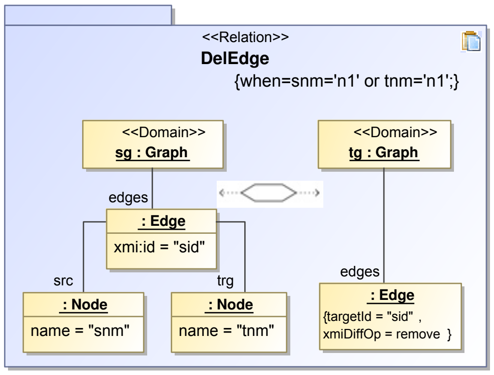


Figure 17: Delete the incident edges

This task is accomplished by an *in-place* transformation, which is defined in QVTR-XSLT by modifications (insert, remove, replace) of the existing model elements. The metamodel has been already defined in Figure 6. The transformation consists of two relations shown in Figure 16 and Figure 17.

The top-level relation `DelNode` is the starting point of the transformation. Its source domain pattern (left part) matches a `Node` named “`n1`” in a graph. If the “n1” node is found, its identifier (`xmi:id`) is bound to variable `sid`, and then the target domain pattern (right part) marks the `xmiDiffOp` of the node whose id is the `sid` as `remove`. Invoked from the `where`-clause of relation `DelNode`, the source domain pattern of relation `DelEdge`, along with its `when`-clause, is used to find all the `Edges` whose `src` or `trg` nodes are the “n1” node. Similarly, these edges are marked as `remove` in the target domain pattern.

An XSLT program of about 80 lines of code is generated for the transformation. Execution of the program copies all the model elements from the input model to the output model, except the ones marked as `remove`.

4.11.2. What is the tool suited for and why?

The tool can be applied in various transformation scenarios, for examples, the platform independent model (PIM) to platform specific models (PSM) transformations, or the transformations of UML models to models of formal languages (such as CSP). Data transformation in data engineering is another potential application field for the tool. Using the tool, the structures of the XML data can be described concisely by the metamodels, the mappings between the data could be specified by relations of QVT-R using the graphical notation. In addition, the tool can be used in the fields of semantic web and ontologies, often there is a need to convert between different knowledge models, which are also in XML formats.

The QVTR-XSLT tool has been successfully used to design complicate transformations that work within CASE tools, one transformation may include more than 100 rules and queries, and generate more than ten thousands lines of XSLT code.

4.12. UML-RSDS

UML-RSDS (Reactive system design support) is a general-purpose language and tool for model-based development. The language is a precise subset of UML and OCL, in which software systems can be specified and designed using UML class diagrams, use cases, state machines and activities. Executable code (in Java) is automatically synthesized from the system design.

In UML-RSDS a transformation specification is expressed by:

1. A class diagram, showing the source and target metamodels of the transformation.

2. A use case, defining the transformation effect. A use case can have a precondition, defining assumptions made about the source and target models at the start of the transformation. It also has a postcondition, defining the intended state of the source and target models at the end of the transformation.

Source and target metamodels are defined using the visual class diagram editor of UML-RSDS.

4.12.1. Delete Node with Specific Name and its Incident Edges

The named node deletion transformation on graphs is an update-in-place transformation, which operates on models of the metamodel shown in Figure 18. This transformation has the following two postconditions:

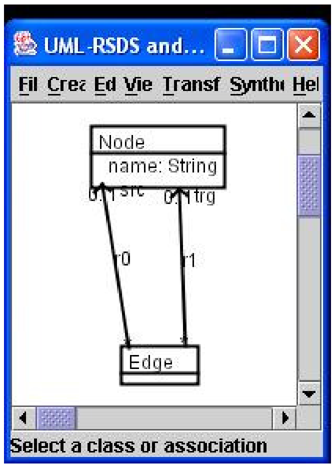

UML-RSDS and...
Fil Crea Ed Vie Transf Synth Hel
Node
name: String
0..1 src
0..1 trg
r0
r1
*
*
Edge
Select a class or association

Figure 18: Graph metamodel in UML-RSDS

```
n:src.name or n:trg.name  implies  self-->isDeleted()
```

operating on instances of `Edge` (the notation `n:src.name` abbreviates `src.name-->includes(n)`), and

```
name = n1  implies  self-->isDeleted()
```

on instances of `Node`, where `x-->isDeleted()` expresses that `x` is removed from the model.

Logically, these postconditions can be read as expressing that in the end state, there are no edges that have a source or target node with name `n1`, and that all such nodes have also been removed. Operationally, the constraints define two transformation rules that remove the edges first, then remove the nodes. We need to remove the edges first, in order to avoid introducing dangling edges without target or source nodes during the transformation.

4.12.2. What is the tool suited for and why?

UML-RSDS has been successfully used for all categories of transformation, except text-to-model. It is particularly suited for update-in-place transformations (refactoring, restructuring, etc.), and for refinements and abstractions. It can be used for migration transformations, e.g., [48], however it does not have specific support for defining migrations. Similarly it does not have specific support for model-to-text or model-merging transformations, but these can be defined. It supports the definition and instantiation of generic transformations.

UML-RSDS has the advantage of using standard UML and OCL notations to specify transformations, reducing the cost of learning a special-purpose transformation language. It also has the advantage of making explicit all assumptions on models and providing global specifications of transformations, independent of specific rules. Verification support is provided for proving transformation correctness [50]. For the above example, we can prove the logical postcondition that exactly those nodes with name $n1$ have been removed using these techniques.

The generated executable implementations of transformations have high efficiency and are capable of processing large models of over 500,000 elements [49].

4.13. Viatra2

The objective of the Viatra2 (VIsual Automated model TRAnsformations [83]) framework is to support the entire life-cycle of model transformations consisting of specification, design, execution, validation and maintenance. Viatra2 uses the VPM (Visual and Precise Metamodeling) approach [81] that supports arbitrary metalevels in the model space. It combines graph transformation and abstract state machines (ASM) [7] into a single framework for capturing transformations within and between modeling languages [80]. Transformations are executed using an interpreter and both (i) *local-search-based* (LS) and (ii) *incremental pattern matching* (INC) are available, providing additional opportunities to fine-tune the transformation either for faster execution (INC) or lower memory consumption (LS) [35].

4.13.1. Delete Node with Specific Name and its Incident Edges

Model transformations written in Viatra2 consist of graph pattern definitions and both graph transformation rules and ASM rules. The `delete-N1NodeAndAllIncidentEdges` transformation below uses graph patterns for identifying the node with the specific name (`N1Node`) and the edges connected to a given node (`connectedEdge`). The example is complete and executable on any VPM model space that contains the proper metamodels. Upon execution, the `main` rule is called, which first attempts to find one node with the given name (`choose` semantics) and then the found node is used as an input parameter to find all incident edges in the model (`forall` semantics). Each incident edge is deleted from the model and finally, the node itself is removed as well.

```
import datatypes; // imported parts of the model-space are usable by local name
import nemf.packages;
import nemf.ecore.datatypes;

@incremental // uses incremental pattern-matcher
machine deleteN1NodeAndAllIncidentEdges{
  pattern N1Node(Node) = {// finds Node with name "n1"
    graph1.Node(Node); EString(Name); // type constraints
    graph1.Node.name(NameRel,Node,Name); // relation constraint
    check(value(Name) == "n1");} // attribute value constraint

  pattern connectedEdge(Node,Edge) = {// Edge is connected to Node
    graph1.Node(Node); graph1.Edge(Edge);
    graph1.Edge.src(SourceRelation,Edge,Node);
  } or { // Node can be source or target of Edge
    graph1.Node(Node); graph1.Edge(Edge);
    graph1.Edge.trg(TargetRelation,Edge,Node);}

  rule main() = seq{ // transformation entry point
    try choose N1 with find N1Node(N1) do seq{ // selects one match
      forall Edge with find connectedEdge(N1,Edge) do // iterates on all matches
        delete(Edge); // delete model element
      delete(N1);}}}
```

Listing 3: Delete node transformation

4.13.2. What is the tool suited for and why?

As a direct consequence of the metamodeling approach of VIATRA2, models taken from conceptually different domains (and/or technological spaces) can be easily integrated into the VPM model space. The flexibility of VPM is demonstrated by a large number of already existing model importers accepting the models of different BPM formalisms, UML models of various tools, XSD descriptions, and EMF models.

The VIATRA2 transformation framework has been applied to a wide variety of problems and its interpreted transformation language, incremental pattern matcher engine and transactional model manager provide a solid foundation for a wide range of applications. The change notifications offered by INC are used to drive a trigger engine to create live transformations [60] where rules are executed in response to specific changes and change driven transformations [4] that translate changes on an input model to changes on output models. The transformation engine also supports interactive execution of rules to drive simulations [61] or perform design-space exploration [23]. Finally, the development of transformations in VIATRA2 are supported by customizable model space visualization [24] and dynamic transformation program slicing [77].

Although VIATRA2 is capable of importing and transforming EMF models, its performance and the conciseness of EMF transformation programs are lower than those of native EMF tools. This is mainly caused by the inefficient importer and the VPM approach that requires a large number of model elements to represent EMF models.

5. Transformation Tool Contest and SHARE

Let us briefly introduce the workshop format of the Transformation Tool Contest (TTC)[7], the hosting event of the Hello World case and the organizational umbrella for this comparison. The aim of the TTC series is to compare the expressiveness, the usability and the performance of graph and model transformation tools along a number of selected case studies. Participants of this workshop want to learn about the pros and cons of each transformation tool considering different applications; and of course how competitive their tool is. A deeper understanding of the relative merits of different tool features helps to further improve graph and model transformation tools and indicate open problems. From the perspective of tool vendors, it gives input on the features to be developed next. The workshop comprises several so-called offline cases, which have to be solved by the participants before the workshop, but also a live case not known in advance that has to be solved by the participants during the workshop. The live case, which is not covered in this paper, allows to evaluate how well-suited the transformation tools are for rapid prototyping.

The review process of the TTC was based on SHARE (Sharing Hosted Autonomous Research Environments), which is described by its creators [20] as: "SHARE is a web portal that enables academics to create, share, and access remote virtual machines that can be cited from research papers. By deploying in SHARE a copy of the required operating system as well as all the relevant software and data, authors can make a conventional paper fully reproducible and interactive."

5.1. Offline solutions workflow

We describe the workflow regarding the offline solutions in more detail, because they allow to understand the role of SHARE.

[7]http://planet-mde.org/ttc2011/

Solving the cases and submitting the solutions: The offline cases were published several months before the workshop. Potential participants solved them with their favorite tools, and submitted their solutions, with a document describing the solution, but especially with an installation in a remote virtual machine offered by SHARE.

Solution review: This SHARE image together with the accompanying paper were then the basis for the peer review. Some time ahead of the workshop, two reviewers also called opponents were chosen by the organizers from the set of all participants to investigate a given solution in detail, and to give a first vote on the solution needed for accepting it to the workshop. The other participants had equal rights to access the SHARE images to inform themselves about the competing solutions. To emphasize it: the very task of the opponents was to find out whether the solutions available in the share images were adequately described by the accompanying papers, whether ugly things were swept under the carpet, or even false claims were made – which can easily happen with traditional, paper-only-based reviews.

Solution presentation: During the workshop, the solution submitters presented their work in front of all other participants. They were able to focus on the strong points of the solutions. But after the presentation a discussion was scheduled, in which the general audience could ask questions, and especially the opponents were on duty to report about the weak points of the solutions, thus balancing the presentation of the solution submitter.

Solution voting: The presentation and discussion were then followed by the voting: All the contest participants were asked to fill out an evaluation sheet for each solution (except their own). The criteria to be used in the evaluation sheet were defined by the case submitters, the participants had to score each solution with 1–5 points regarding each criterion. The solutions were ranked and awarded prices along the votes of the participants.

5.2. Cases

There were 4 offline cases to be solved, “Program Understanding” [32], “Compiler Optimization” [8], “Model Migration” [26], and the already introduced “Hello World!” [55] case.

The complex cases Program Understanding, Compiler Optimization, and Model Migration complement the Hello World case with non-primitive tasks. They allow to assess the ability of the tools to cope with difficult tasks and large workloads, evaluating expressiveness and performance. This gives hints on the scalability of the tools when facing complex tasks and real world workloads. A very short introduction into the results of these tasks is given in the following section, for the detailed results we must hint at the proceedings of the TTC [78].

6. Votes and Discussion

As described in the previous chapter, the solutions were i) investigated in detail by the opponents and ii) voted by the participants at the transformation tool contest. Here, we publish the votes for the Hello World case, which were cast along the dimensions completeness, understandability, and conciseness in steps of 1 to 5 points, with 5 being the highest score. Additionally, we give an interpretation and discussion comparing the solutions and tools in the order of voting results. The discussion is based on the opponents statements, enriching them with further post-workshop insights of the authors. Only tools that were presented at the TTC were voted – tools that were not voted are given at the end. Only tools that were submitted in time were reviewed by opponents – tools that were not reviewed are not discussed.

Completeness was of low impact compared to conciseness and understandability, with the worst solution in this regard scoring at 95% of the maximum value (compared to 56% and 57% regarding the other dimensions). This high rate of success is not surprising taking into account how basic the tasks were; in fact it is rather surprising that a third of the tools was not able to give a complete/correct solution in the first place. So the matter was decided alongside understandability and conciseness. Regarding understandability, the distinction into graphical versus textual played a role, complemented by the concepts the tool is built on (e.g., constructs from formal logics received a malus), and whether the tool offers a syntax similar to other well-known languages. Regarding conciseness, the availability of lightweight means for simple CRUD tasks played a role, but even more so the general expressiveness of the tools, as expressed by the availability of the features referenced in the feature matrices; they had not to be used into great depth, but their general

Table 12: Outcome of the voting for the tool solutions

Criteria	**GReTL**	**UML-RSDS**	**MDE Lab**	**Groove**	**EMF Henshin**	**Epsilon**	**Edapt**	**MOLA**	**Via-tra2**	**GrGen .NET**	**PETE**
Compl.	5,00	4,83	4,77	5,00	5,00	4,93	5,00	4,86	5,00	5,00	5,00
Underst.	2,92	2,85	4,08	3,67	3,69	4,33	3,47	4,00	3,86	3,77	2,92
Concisen.	4,23	3,54	3,46	4,54	3,38	4,33	3,47	3,57	2,79	3,77	3,62
Average	4,05	3,74	4,10	4,40	4,03	4,53	3,98	4,14	3,88	4,18	3,85
Rank	6	11	5	2	7	**1**	8	4	9	3	10

availability already lead to more compact solutions compared to competing tools.

6.1. Epsilon

The Epsilon solution was able to employ a language matching the task at hand for nearly each task of the Hello World case. So the solution scored highly regarding conciseness and understandability, rendering it the clear winner of this challenge. But there is one exception to the understandability and conciseness: as no declarative pattern matching is offered, the match a cycle of nodes task had to be coded imperatively with 3 nested loops. Offering multiple languages has another side effect, which was also complained about by the opponents: learning becomes harder.

6.2. GROOVE

GROOVE was able to solve each task with one rule, which resulted in the highest conciseness vote of all competing solutions. This was made possible by the use of quantified nodes – but while yielding the top vote regarding conciseness, they lead to only a midfield place and complaints from the reviewers regarding understandability. A further point criticized was the lack of XMI import and export. *(Note that this last point has been addressed since; the newest version offers a versatile import and export to Eclipse/EMF (meta)models.)*

6.3. GrGen.NET

GrGen.NET was as balanced as Epsilon regarding conciseness and understandability, but at a clearly lower level. Strong points regarding conciseness were the retyping for the migration task, which earned it kudos from the opponents, and the subpattern iteration. Complaints were fielded regarding the lack of an exporter for XMI, which was manually coded, and an imperatively formulated transitive edges solution.

6.4. MOLA

The MOLA solution received the 3rd highest vote regarding understandability, but only a midfield result regarding conciseness. It can be expected that the graphical language for both rules and control leads to the high vote for understandability while reducing the conciseness score.

6.5. MDELab SDI

The MDELab SDI solution received the 2nd highest vote regarding understandability, which most likely must be attributed to its clean graphical syntax. But this was offset by the 3rd lowest vote regarding conciseness, with the opponents complaining about the lack of non-isomorphic matching and the unavailability of negative patterns, which would have been useful for this case.

6.6. GReTL

The GReTL solution scored 3rd highest regarding conciseness based on the highly compact language for the transformation of query result sets. But on the other hand it scored 2nd worst regarding understandability, which must be attributed to a lot of special tokens not known from common programming languages, which seem to stem from formal specification languages the audience was not trained in.

6.7. Henshin

Henshin scored in the midfield for understandability, which is rather surprising for a graphical rule-based language, especially compared to MDELab SDI and MOLA – it seems that the non-activity-diagram-based control language gave a minus in this regard. It scored 2nd worst regarding conciseness, with an opponent criticizing it for the lack of a pure apply-rule-for-all-matches construct not requiring a kernel rule.

6.8. Edapt

Edapt offered a strong solution for the model migration task for which the tool is designed, only reusing some predefined operations. For the other tasks, hand-coded solutions against an API were offered, which gave complaints by the opponents regarding a lack of declarativeness and conciseness and a reduced vote in this areas.

6.9. VIATRA2

VIATRA2 was seen by the voters to belong to the top third regarding understandability, but received a devastating vote regarding conciseness. While clearly being one of the most verbose languages, the effect seems disproportional, which most likely is to be attributed to the solution introduction paper that presented several possible solutions for each of the tasks; while being in accordance with the goals of the Hello World case it seems that people just glimpsing over paper became the impression that there is a huge amount of code needed.

6.10. PETE

Pete received kudos for its declarative and potentially bidirectional style for specifying transformations with Prolog terms. On the other hand, this most likely gave it a bad vote regarding understandability with an audience not trained in this language. In ended in the midfield regarding conciseness, with an opponent praising it for a concise solutions regarding the counting task and denouncing it for a verbose solution regarding the model-to-text task.

6.11. UML-RSDS

UML-RSDS scored in the midfield regarding conciseness but worst regarding understandability. The opponents termed the solution interesting regarding its basic approach while at the same time asking about the use of mathematical formulas for the simple Hello World task; it seems the voters did not appreciate the predicate logic style of specification either.

6.12. Validity and full solutions

The Hello World task descriptions, despite being simple, contain several ugly corner cases, e.g., reversing dangling edges. Some tool vendors delivered a correct solution offering exactly the required features, at the price of a more complex and thus less concise and readable solution, while others were flexible in the interpretation of the requirements, concentrating on the pragmatics of the case, i.e., introducing their tools. Some voters took this into account and gave lower votes regarding completeness based on correctness, while others just ignored this issue or were even surprised that some participants voted them in this regard.

The votes cast can be seen as good indication on the overall performance of the tools regarding understandability and conciseness, but should be taken with a grain of salt. On the one hand, they are tweaked by the presentational skill of the tool vendor before the audience, and on the other by the inquisitional skill of the opponents. For while in principle every tool vendor could evaluate all tools and then give an all encompassing vote, this rarely happened with more than a dozen competing tools and solutions; the SHARE

Subject	**SHARE**	**Cases solved**
ATL/EMFTVM	[85]	Hello World
Epsilon	[64]	Hello World
Edapt	[31]	Hello World, Prog. Understanding, Model Mig.
GReTL	[33]	Hello World, Prog. Understanding, Compiler Opt.
GrGen.NET	[14]	Hello World, Prog. Understanding, Compiler Opt.
GROOVE	[62]	Hello World, Compiler Opt.
Henshin	[44]	Hello World, Prog. Understanding
MDELab SDI	[89]	Hello World, Prog. Understanding
metatools	[51]	Compiler Opt.
MOLA	[39]	Hello World, Prog. Understanding
QVTR-XSLT	[11]	Hello World, Compiler Opt.
UML-RSDS	[47]	Hello World, Model Mig.
VIATRA2	[22]	Hello World, Prog. Understanding

Table 13: SHARE images and solved cases of the tools

images of the solutions typically were investigated by the opponents and a handful of interested, but not all participants.

Regarding understandability one should take care of the fact that this criterion is highly dependent on the experience and knowledge of the voters. Tools offering a notation based on general programming constructs fared better than tools based on constructs from formal specification languages or formal logic. While this gives a good indication on the mass market compatibility of a tool and the appeal to the average software engineer, it might give a wrong impression on the understandability of a tool regarding the knowledge available to a prospective user.

Conciseness was judged subjectively, too, based on the presentation and the paper. Presenting only the concise parts could have lead to a better vote then deserved regarding overall performance. Some help in this regard and especially regarding the question of correctness might come from a transformation judge [56], an objective computer program under development.

Given all the subjective influences on the voting, we want to encourage any prospective user to have a look on its own: All tools and their solutions to the Hello World case (as well as to the other cases of the TTC) are available in the SHARE images listed in Table 13, ready to be reproduced and interpreted.

7. The tools in detail

In the following, we work again based on the setup we already employed in Section 2, asking typical questions of prospective tool users, which are then answered with explanations and feature matrices. Here we refine the initial answers that were geared towards giving an overview of the field with a step into greater detail. The focus in this section is on comparison and especially filtering alongside questions a user who needs to decide in between tools typically raises, or would raise if their importance was known. The features needed for answering these questions are sometimes very different from those typically found in scientific transformation tool comparisons of experts, and they are often answered at a lower level of detail than commonly found.

The tool comparison is organized along the areas which were already used in Section 2, namely:

Data: Which data is to be transformed?

Computations: What kinds of computations are available, how are they organized?

Languages and user interface: How does the interface of the tool to the user look like?

Environment and execution: How does the interface of the tool to the environment look like, how is it executed?

Suitability was already deepened in Section 4, and is thus not taken care of any more in its direct high-level form – but it is very well taken care of indirectly by the detailed features, which were chosen to allow to filter out tools not suitable to the task at hand quickly. In addition to the areas named, the support for **Validation and Verification** is investigated in more detail. It is explained why and when the selected features are of importance.

7.1. The data refined

"Can I adequately model my domain?" received only a very coarse grain answer in Subsection 2.2, but is a question of high importance, as the model comprises the foundation on which all of the computations and all of the other tool features are built. So we have to investigate it more deeply, regarding the expressiveness of the metamodel and the import/export capabilities of the tool.

7.1.1. Input and Output

"Does the tool support the file formats I need?" is the question answered in the following Table 14, which compares the tools regarding their import and export capabilities. A tool built on its own modeling technology may still be able to import and export the serialization format of another modeling technology by mapping those concepts to its own concepts. Beside allowing to remove tools from the set of candidates that do not offer the format one needs, this information gives a means of measuring tool interoperability.

The available formats listed below are:

GXL: an XML dialect, the de facto standard for graph transformation tools, specifying the graph as well as the type graph.

XMI: an XML dialect, the de facto standard for model transformation tools, specifying the model. The metamodel for an .xmi is typically given in .ecore format (version 2.0).

XMI(1): in case the tool supports XMI 1.x.

XML: general support for XML.

CST: custom formats, which are not helpful in tool interoperability (unless adopted by other tools thus defining a de facto standard), but unleash the full potential of a tool.

TXT: means the tool has support for text output into a file, which allows to emulate an arbitrary file format (at the price of explicit coding). If this is specified for input, the tool offers integration with a parser generator, e.g., EMFText.

VIZ: visualization formats emitted by the tool, which are used as input for a graph visualization tool (e.g., .dot or .vcg).

VER: file formats for verification, e.g., .aut.

7.1.2. Metamodel Expressiveness

For the following parts we are back again at the original question "Can I adequately model my domain?", now having a look in greater detail at what the tools are able to express, i.e., what the metamodels support. First we respond to the query "Do the nodes and edges allow me to directly encode my problem?", with Table 15.

Tool	**Input formats**	**Output formats**
ATL/EMFTVM	XMI, TXT	XMI, TXT
Epsilon	XMI, XMI(1.x), XML	XMI, XMI(1.x), XML
Edapt	XMI	XMI
GReTL	CST, XMI, GXL	CST, XMI, GXL, VIZ
GrGen.NET	CST, GXL, XMI	CST, GXL, TXT, VIZ
GROOVE	GXL, XMI	GXL, XMI, VIZ, VER
Henshin	XMI	XMI, VIZ, VER
MDELab SDI	XMI	XMI
metatools	TXT, XML(dtd)	TXT, XML(dtd)
MOLA	XMI, CST	XMI, CST, TXT
QVTR-XSLT	XML, XMI, GXL, CST	XML, XMI, GXL, CST
UML-RSDS	CST	CST, XMI
Viatra2	XMI, CST	XMI, CST, TXT

Table 14: Input and Output

Nodes and Edges. There are different ways in which the nodes or edges may be typed:

U: means the elements are untyped.

P: for plain means the elements are equipped with disjoint types.

S: for single inheritance means the elements are equipped with a type hierarchy specifying a can-be-substituted relation with one single parent for each type.

M: for multiple inheritance means the elements are equipped with a type hierarchy specifying a can-be-substituted relation with potentially multiple parents for each type.

Additionally, an element may bear attributes, denoted with **A**.

Modifier. In contrast to the nodes that are equipped with very few degrees of freedom, there are many differences in how edges (resp. references) are exactly realized.

+M: If the edges have an identity, i.e., multiple edges of the same type between two nodes are allowed, we speak of a Multi-Graph; we note down +M in this case. This is an extension of a simple graph, which allows only one edge of the same type in between two nodes. The simple graph is the natural choice for problems, which are strictly built on relations, where multiple edges between same nodes do not have any meaning.

+U: Edges are normally directed, with one node distinguished as source and the other as target. There might be additionally undirected edges. In this case, we hint at them by noting down +U.

+O: Edges in a graph are typically unordered (in contrast to an array of references). In case the tool is able to query for and assign the position of edges, this is denoted by +O for ordered.

+RC: Model transformation tools typically distinguish edges into references and containment, and e.g., implement a cascading delete of children if the parent is deleted. It is indicated by +RC that this is the case.

+I: The tool supports inverse edges if creating or deleting an edge automatically creates or deletes a co-edge in the opposite direction.

+B: The tools supports traversing edges in both directions (bi-directional navigationability). This is common for graph tools, the co-edge handling can be seen as a way to achieve this behavior in a model-based tool.

After our look at the elements defining the structure, we now examine the attribution more closely, responding to the query "Are the attributes available that I need to directly encode my problem?" with Table 16.

Model Kind. The graph elements may be attributed freely, or depending on their type.

Liberal: means each element can bear arbitrary attributes.

Fixed: means a type defines uniquely which attributes an element may bear.

The liberal kind is more flexible, but rules out static type checking.

Tool	**Nodes**	**Edges**	**Modifier**
ATL/EMFTVM	M, A	P	+M, +O, +RC, +I
Epsilon	S[1]	[1]	+M, +O[1]
Edapt	M, A	P	+M, +O, +RC, +I
GReTL	M, A	M, A	+M, +O, +RC, +B
GrGen.NET	M, A	M, A	+M, +U, +B
GROOVE	M, A	S	+B
Henshin	M, A	P	+RC
MDELab SDI	M, A	P	+M, +U, +O, +RC, +I
metatools	M, A		+M, +O
MOLA	M, A	P	+U, +O, +RC, +I
QVTR-XSLT	M, A	P	+M, +O, +RC
UML-RSDS	S, A	P	+O, +RC, +I
VIATRA2	M, A	M	+M, +RC

[1] depends on the underlying modeling technology, values given for EMF

Table 15: Nodes and Edges

Attributes. The following table defines which types are available for typing attributes, potentially available are: **b** for boolean, **i** for integer, **f** for floating point, **s** for string, **e** for enum, **o** for object (user-defined types, which are opaque to the tool language, they make only sense when the tool can be employed from an API embedded in a host program), **C** for collection types (e.g., set, map, bag, array, list).

We complete the model part of the comparison by stating that all the computations compared next operate on an in-memory-representation, in contrast to databases (e.g., graph databases).

7.2. The computations refined

"Can I adequately specify my computations?" was answered in Subsection 2.3 by explaining the different approaches and their expressiveness regarding collecting data, especially in respect to depth and breadth splitting structures. The latter aspects gave already an explanation of expressiveness at some deeper level of detail, we continue there later on, but now we begin with an inspection of some software engineering aspects.

7.2.1. Programming in the large and Reuse

"Is the tool suited to a complicated transformation task?" is a question that is answered to a good degree by the points regarding expressiveness we

Tool	Model Kind	Attributes						
		b	i	f	s	e	o	C
ATL/EMFTVM	Fixed	✔	✔	✔	✔	✔	✔	✔
Epsilon	Fixed	✔	✔	✔	✔	✔	×	✔
Edapt	Fixed	✔	✔	✔	✔	✔	×	✔
GReTL	Fixed	✔	✔	✔	✔	✔	×	✔
GrGen.NET	Fixed	✔	✔	✔	✔	✔	✔	✔
GROOVE	Fixed or liberal[1]	✔	✔	✔	✔	×	×	×
Henshin	Fixed	✔	✔	✔	✔	✔	✔	×
MDELab SDI	Fixed	✔	✔	✔	✔	✔	✔	✔
metatools	Fixed, Liberal	✔	✔	✔	✔	✔	✔	✔
MOLA	Fixed	✔	✔	×	✔	✔	×	×
QVTR-XSLT	Fixed	✔	✔	✔	✔	✔	×	×
UML-RSDS	Fixed	✔	✔	✔	✔	✔	×	×
VIATRA2	Liberal	✔	✔	✔	✔	✔	×	×

[1] The user can either set a type graph fixing the allowed attributes, or leave the model untyped

Table 16: Attributation

already visited (and will visit), but additionally by the question "What means of abstraction and reuse are available?" and "What means of structuring large specifications are available?". In the following Table 17, the capabilities of the tools for transformation programming in the large are listed. Of course it depends on the size of the task at hand whether or to what extend this abilities are really needed.

Model and Computation structuring. Is it possible to combine the metamodel specification in a project from parts in different files or views, which can be reused and edited separately? And as a refinement: do they support different namespaces, so that the parts can be combined without name clashes? A check mark for the first questions allows for metamodel structuring and reuse, the refinement is important if models out of different origins need to be combined.

Is it possible to combine the computations in a project from parts in different files or views, which can be reused and edited separately? And as a refinement: do they support different namespaces, so that the parts can be combined without name clashes? A check mark for the first questions allows for computation structuring and reuse, the refinement is important if computations from different programmers need to be combined.

Abstraction and Parameterization. The most basic unit of reuse in the world of transformation are entire transformations between representations, which are combined by transformation concatenation, building a pipeline architecture. Transformations that would be too complex for one pass are split into multiple passes linked by intermediate representations.

Besides this external reuse one needs to ask in how far the units from which the transformations are built can be reused: When transformation elements are abstracted into own units, what are these units, and how can they be parameterized? We concentrate on what a subprogram of the program-based approach would offer, just split into the different units employed by the approaches (as induced by their layering).

In the program-based approach, a program is the basic unit of use. If it is in addition possible to factor out common program computations and reuse them from other program parts, we note down **P** for a subprogram. The subprogram subsumes all the other means of reuse offered by alternative approaches.

In the library-based approach, library functions are the basic unit of use. As it is not possible to abstract them further we omit them here.

In the query-update-based approach, a query with update is the basic unit of use. When it is in addition possible to factor out common queries and reuse them from other queries, we note down **Q** for subqueries.

In the rule-based approach, a rule is the basic unit of use. The explicit control sequence or an implicit control engine is an accompanying basic unit of use. When it is in addition possible to factor out a control sequence of a rule control language (or a program when the rule control language includes a programming language), and reuse that unit as a compound control unit, we note down **C** for a sub-control program. When it is in addition possible to factor out a rule and reuse it from another rule, we note down **R** for a subrule. When is it in addition possible to factor out a computation below rule level and reuse it from another rule, we note down **S** for subrule computations; this includes e.g., application conditions and subpatterns. **C** is helpful when complicated transformations need to be programmed. **R** saves one from returning output parameters to the control language to be used as input parameters for following rules for tools of the rule and explicit control approach, which is a cumbersome practice, and allows implicit control

Tool	**Mod.**	**Com.**	**Units of abstraction and reuse**
ATL/EMFTVM	/	/	C(I,O), R(I,O), S(I,O)
Epsilon OL	/	/	P
Epsilon TL	/	/	R, S
GReTL	/	/	Q(I,O), P(I,O)
GrGen.NET	✓/×	✓/×	C(I,O), R(I,O), S(I,O)
GROOVE	/	/	C, R(I,O)
Henshin	/	/	C(I,O), R(I,O)
MDELab SDI	✓/✓	✓/✓	C(I,O), R(I,O), S(I,O)
metatools	partly	/	P(I,O)
MOLA	✓/✓	✓/×	C(I,O), R(I,O)
QVTR-XSLT	/	/	R(I,O)
UML-RSDS	/	/	C(I), R(I)
VIATRA2	✓/✓	✓/✓	C(I,O), R(I,O), S(I,O)

Table 17: Transformation organization and reuse

tools to use other rules as a reusable entity. **S** is helpful when there are common structures to be processed in the same way or non-trivial attribute computations to be carried out from several rules.

Do these parts support parameterization?

I if parameters passed in are supported.

O if parameters passed out after applying the part are supported.

7.2.2. Extensibility, runtime flexibility, and meta programming

"I have a subproblem the tool does not allow me to solve, what can I do?" is a question which arises if the tool is not used in a pure transformation setting (for which the tool and especially its languages were designed for). The tools introduced in this article for example typically do not natively support matrix or vector classes in the model and matrix-vector-multiplications in the attribute computations, or the filtering of matches by additional queries against a persistent database before they are applied. The extensibility of the languages and the tools are visited in Table 18, besides their flexibility concerning runtime changes, and their support for meta-programming.

Extensions. Is it possible to extend the transformation with external program code? In which way, and with what language?

EOP Externally-defined transformation operations can be called in place of the built-in operations, or in place of operations specified in the language of the tool.

EAC Externally-defined attribute types and attribute computations can be defined; e.g., for introducing a type matrix and a matrix multiplication as operation, which are opaque to the tool.

Externally-defined transformation operations are helpful when the attributes offered by the tool as given in Table 16 and the operation available on them are not sufficient for the task at hand. Externally-defined transformation operations allow to fall back to a programmed model for tasks where this is more appropriate; in contrast to an API which is used from outside the tool languages, they allow to utilize programmed operations from within the tool, e.g., calling an external operation from the rule control language of the tool.

Runtime adaptability. "Can I adapt my metamodel or my computations at runtime?" is a question that is rather seldom raised, but if it is raised it reduces the set of potential candidates by a wide margin. The capability to change the metamodel at runtime is shown with a check mark in the Model column. In this case, a target model can be built when executing a transformation. The capability to change the computation at runtime is shown with a check mark in the Comp. column. In this case and for a rule based language, rules from the rule set can be changed while the transformation is running, depending on intermediate results.

Meta-Programming. "Can I operate at the meta level?" is a question whose answer gives an indicator of the conciseness that can be achieved with the tool; the practical question behind it is "Can I escape boilerplate code?" for tasks that require repetitive specifications.

A meta-iteration allows to iterate over all available types in the model, not naming the specific types. Tasks where a lot of types are to be treated in the same way can be specified concisely in a loop over the types, instead of being explicitly and statically enumerated in the code. The capability to program at this level is denoted with a check mark in the Meta-I column.

When the computations are available as models to be inspected and changed, the model transformation tools introduced here are able to synthesize the computations by a meta program in the tool itself. Tasks where

Tool	**Extensions**	**Model**	**Comp.**	**Meta-I**	**Meta-P**
ATL/EMFTVM	EOP(Java), EAC(Java)	×	✓	✓[1]	✓
Epsilon	EOP(Java)	✓	×	✓	×
Edapt	EOP(Java)	✓	✓	✓	×
GReTL	EOP(Java)	✓	✓	✓	×
GrGen.NET	EOP(C#), EAC(C#)	×	×	×	×
GROOVE	-	×	×	✓	✓
Henshin	EAC(JavaScript)	✓	✓	×	✓
MDELab SDI	EOP(Java), EAC(Java)	×	✓	×	✓
metatools	Comp. *are* external	×	✓	✓	✓
MOLA	EOP(C++, Java)	×	×	×	✓
QVTR-XSLT	EOP(XSLT)	×	×	×	✓
UML-RSDS	-	×	×	×	×
VIATRA2	EOP(Java), EAC(Java)	✓	×	✓	✓

[1] only supported by the rule control language

Table 18: Extensions, runtime flexibility, and metaprogramming support

a lot of similar computations arise can be specified concisely with a meta-program (at the price of understandability); we denote this capability with a check mark in the Meta-P column.

Tools that do not offer these capabilities require either copy-n-paste programming or to program a tool-external code generator, which emits specifications in the tool's format.

7.2.3. Expressiveness in detail

"Does the tool allow me to adequately specify my computations?" is a question that was already visited in the Subsection 2.3. In addition to the introduction into the field and the medium level discussion of depth and breadth matching given there, we want to examine here some more detailed points in Table 19; some of them were of importance for the Hello World tasks and played a role in the reviewer comments and the votes.

Retyping. Retyping, also known as relabeling in graph rewriting, allows for an in-place change of the type of a node while keeping its incident edges. We note down **cross** in case it is supported without restrictions, **down** if only downcasts are allowed (being type safe but less powerful), and otherwise **none**. This feature allows rewrite tools to achieve concise solutions for simple 1:1 mapping tasks, it especially allows pattern-based tools with their weakness in describing a statically not fixed context to achieve concise solution.

Transaction support. Transactions allow to roll back changes carried out on the model at request. We note down **simple** if they are supported, **nested** if they may be nested, **debugger** if they are only available in the debugger to interactively try out things, and **none** otherwise. Transactions render programming for search-based tasks easier, as it is possible to just try a transformation and to roll it back to the original state when it failed according to some criteria, instead of being forced to specify a complex condition that needs to be checked beforehand, or instead of being forced to explicitly undo the effects. Transactions are only important for tasks where one must search for an optimal model, for plain transformation tasks they are not needed.

Modifier Matching. The matching modifiers constrain the way in which pattern elements may be matched to graph elements; they only apply to pattern-based tools. We write:

iso if a single graph element cannot get matched by multiple pattern elements.

hom if a graph element can get matched by multiple pattern elements.

The default is specified first. Tools that only offer isomorphic matching must duplicate patterns for tasks where non-isomorphic matching is needed. Tools that only offer homomorphic matching require the reader to always think of all the possibilities in which pattern elements may get coalesced by matching them to the same graph element.

Modifier Rewriting. The rewriting modifiers constrain deleting nodes in case edges not mentioned in the pattern would dangle; they only apply to pattern-based tools. We write:

SPO if the rule is applied and the edges that would dangle are deleted.

DPO if the rule is prevented from matching and nothing happens.

The default is given first. SPO allows to delete a node even if not all edges were specified in the pattern, which is for most tasks the more practical approach. With DPO semantics, proving propositions about graph rewrite

Tool	**Retype**	**Transaction**	**Mod. Mat.**	**Mod. Rew.**
ATL/EMFTVM	none	none	iso, hom	DPO
Epsilon	cross[1]	simple	-	-
Edapt	cross	none	-	-
GReTL	none	none	-	-
GrGen.NET	cross	nested	iso, hom	SPO, DPO
GROOVE	down	none	hom, iso	SPO
Henshin	none	simple	iso, hom	SPO, DPO
MDELab SDI	none	none[3]	iso	SPO
metatools	none	none	-	-
MOLA	none	none	hom, iso	SPO
QVTR-XSLT	none	none	hom	SPO
UML-RSDS	none	none	hom, iso	SPO
VIATRA2	cross[2]	nested	iso, hom	SPO

[1] depends on the underlying modeling technology, values given for EMF
[2] only supported by the rule control language
[3] transactions are supported in the debugger

Table 19: Retyping, transactions, and pattern modifiers

systems is much easier (this is of interest for verification). This is another occurrence of the inability of the pattern-based tools to describe a statically not fixed context.

7.3. The languages and the user interface refined

"Does the user interface of the tool fit to my needs or preferences?" was answered in Subsection 2.4 by listing the languages offered by the tools and a distinction of their form into textual and graphical. Here we refine these points by comparing the support offered by the tools in developing transformations in Table 20, and by a comparison of how easy it is to get acquainted with the tool and its languages.

7.3.1. Development support

"What kind of support do I get by the tool when developing transformations?" is the question to be answered by the following table. Domain specific programming languages *as such* are better suited to transformation problems, and the typically offered visual style of programming and debugging is often more adequate for transformation tasks, but they commonly fall short in advanced editing support like auto-completion or refactorings offered by the IDEs of general-purpose programming languages.

IDE and editor integration. “Can I use my favorite Integrated Development Environment or my favorite editor to develop my transformations?” The IDEs for which a tool integration is available are given, an editor integration must consist of at least syntax highlighting;

Development support. We focus on the abilities to debug and edit one’s transformations. For debugging we are especially interested in the abilities to visualize the model. Regarding editing we query for help in basic editing with auto completion, and for the ability to refactor the specifications, e.g., to consistently rename an entity definition; without refactoring support the changes must be carried out by hand, e.g., for the renaming with a search and replace in a textual editor, fixing the identifiers which were captured accidentally afterwards.

GV Graph viewer – a graph viewer allows to view the graph before the transformation and after the transformation.

HG Hierarchical graph support – the graph viewer allows to view a nested graph, where nodes are capable of containing further graphs. This feature is important for large graphs, which are not consumable by the human mind if presented in a flat layout.

GD Graphical debugger – a graphical debugger allows to execute a transformation stepwise, rule match by rule match, and to view the matching pattern highlighted in the graph.

DB Textual debugger – a debugger for a textual transformation language is available.

CD Change difference – the tool offers a viewer to highlight changes in a model.

RF Refactoring – an editor that allows for some refactorings of the tool language is available.

AC Auto completion – an editor that helps in editing by suggesting completions for the current editing operation is available.

Tool	**IDE & Editor**	**Developer support**						
		GV	HG	GD	DB	CD	RF	AC
ATL/EMFTVM	Eclipse	×	×	✓	✓	×	×	✓
Epsilon	Eclipse	×	×	×	✓	×	×	×
Edapt	Eclipse	✓	×	×	×	✓[3]	✓	×
GReTL	Eclipse, Emacs	✓	×	×	×[2]	×	×	✓
GrGen.NET	N++[1], Vim, Emacs	✓	✓	✓	×	×	×	×
GROOVE		✓	×	✓	×	×	×	✓
Henshin	Eclipse	×	×	×	×	✓[3]	×	×
MDELab SDI	Eclipse	✓	×	✓	×	✓[3]	×	×
metatools		✓	×	×	✓[2]	×	×	×
MOLA	Eclipse	✓	×	×	×	✓[3]	✓	✓
QVTR-XSLT	MagicDraw UML	×	×	×	×	×	×	×
UML-RSDS		×	×	×	×	×	✓	×
VIATRA2	Eclipse	✓	×	×	×	×	×	×

[1] N++ abbreviates the editor Notepad++.
[2] The debugger of the general-purpose programming language is available.
[3] EMFCompare can be used to compare models and visualize differences.

Table 20: Developer support

7.3.2. Learnability

"How easy is it to learn the tool and its languages?" is a question that depends on the quality of the documentation and the already available knowledge.

User documentation. The following types of user documentation may be listed in Table 21: **UM** for user manual, **Tut** for a tutorial, **Wiki** for a wiki, **LR** for a language reference, **API** for an API documentation, **Expl** for examples, **IPHelp** for an in-program help, e.g., Eclipse help, **Pap** for scientific papers, and **Web** for a web site. Numbers appended in parenthesis specify the amount of A4 pages when printed.

The three most important pieces of documentation are given in descending order. They tell a prospective user where to look at first and give a hint on the overall documentation support one can expect.

Direct. Further on learning a tool may be supported by reusing already existing notations. Are notations from standardized programming or specification languages reused in the tool, for attribute evaluation and assignment or application conditions? Maybe the computations are even specified in a general-purpose programming language. If so, which ones?

Learnability. Learning a tool may be fostered by conceptual reuse. So we ask which knowledge from what domains is helpful to understand the tool, reason by analogy? Available are:

FLO Formal logic. For predicates, quantifiers, proofs.

SL Specification languages, e.g., Z, B. For set comprehension and set operations, logic operations, map operations, and their specific syntax.

FLA Formal languages. Grammar rules of nonterminals as placeholders for things yet to be derived/parsed (subpatterns) are employed, and terminals (nodes and edges). Programming with recursion.

DQ Database queries. A query language returning a set of tuples and an update language.

PL Programming languages are characterized by: block nesting, keywords, definition and use of identifiers, statement sequences, expressions for attribute evaluation, and procedure calls.

Tool	**User documentation**	**Direct**	**Concepts**
ATL/EMFTVM	UM, Tut, Wiki	OCL	PL, OO
Epsilon	UM(238), Tut, LR	OCL	PL, OO, RP
Edapt	Tut, API, Expl		PL
GReTL	API, Pap, Expl		SL, DQ, PL, OO
GrGen.NET	UM(250 incl. LR), Expl, API	Java	PL, OO, RP, FLA
GROOVE	UM(25), Tut, Expl		FLO, PL, GL, RP
Henshin	Tut, Wiki, Expl	JavaScript	OO, GL, RP
MDELab SDI	IPHelp, Expl, API	OCL	OO, GL, RP
metatools	Pap, UM(200) API	Java	FLA, PL, SL, OO
MOLA	Web, UM(28), LR(60)	OCL	PL, OO, GL, RP
QVTR-XSLT	Web, Expl, Pap	OCL	GL, RP
UML-RSDS	UM(20), Expl, Pap	OCL,UML	FLO, SL, OO, GL
VIATRA2	Wiki(42), UM(115), LR(88)		PL, RP, DQ

Table 21: User documentation and learnability

OO Attributed classes with inheritance hierarchies over types and method call notation.

GL Graphical languages: UML class diagrams, UML object diagrams, UML activity diagrams.

RP Rule-based programming and pattern matching. Specification by matching of inputs, generation of outputs.

The more proficient the reader is in the domains given by the tool, the easier should be the transformation language to learn. (The less one knows what is given, the harder to learn the tool languages. But tools that conceptually do not build on others are the most expensive to learn, as everything is new.)

7.4. The environment and the execution refined

"In which environment can I use the tool?" and "How are the tool specifications executed?" were already answered in Subsection 2.5. Here we concentrate on answers to the question "What do I have to provide to use the tool?", given in Table 22.

Prerequisites Execution. "What must be installed on a computer besides the bare OS to run developed transformations?" asks about the third-party software an end user is forced to install in order to get a transformation developed

Tool	**Prerequisites Execution**	**Prereq. Development**
ATL/EMFTVM	Eclipse, EMF, ATL core	ADT[1]+ SimpleGT
Epsilon	Java	Eclipse
Edapt	Java	Java, Eclipse (ME)
GReTL	Java	Java
GrGen.NET	.NET 2.0 (or mono)	Java 1.5 RE
GROOVE	Java 1.6 RE	
Henshin	Java, EMF	Eclipse, EMF
MDELab SDI	Java, EMF	Eclipse, EMF
metatools	Java RE	Java SDK, ANTLR, GNU make, coreutils
MOLA	Java, EMF or nothing(C++)	MOLA tool
QVTR-XSLT	XSLT processor	MagicDraw UML
UML-RSDS	Java	Java
VIATRA2	Java	Eclipse

[1] ADT abbreviates the ATL development tools.

Table 22: Prerequisites

with the tool running (copying a few runtime libraries supplied by the tool, which can be shipped with a solution do not count into this).

This is a much stronger requirement than the following prerequisites for the development, as each end user of the transformations is forced to install these prerequisites, or the transformation developer is forced to install them on the end user's computer with a setup routine together with the transformations as such.

Prerequisites Development. "What must be installed on a computer in addition to the execution prerequisites to develop transformations?" asks about the third-party software the transformation developer needs to install in order to develop transformations with the tool (besides the software components directly shipped with the tool).

The prerequisites listed give a hint on the effort required to get the tool running. While a long list for the development prerequisites is not much of a hindrance once the decision for a tool offering compelling features was taken, it is a large obstacle for prospective developers or reviewers in software engineering workshops, which just want to evaluate the tool. The pre-installed SHARE images of the tools are an important help in this case.

7.4.1. License and Maturity

We start with the question “What am I legally allowed to do with the tool?”, and especially “What is the price of the tool?”, but as well as “Is the source code available?”, which are answered in Table 23.

License. The first distinction is whether the source code is available or not.

open the source code is available to the public.

closed the source code is not available, only binaries are supplied.

When the source code is available one can fix bugs and extend the tool on its own, even after the tool run out of support, so this defines a kind of insurance.

The second distinction is along the price, which may be measured in money or in code.

money using the tool requires payment.

strong using the tool requires publishing the using code.

weak changing the tool requires publishing the changes.

free none of the above.

Available open source licenses are:

Apache the Apache License, of type free (mostly).

GPL the GNU General Public License, of type strong.

LGPL the Lesser GPL, of type weak.

EPL the Eclipse Public License, of type weak.

“How mature is the tool?” is an important question with allows to estimate tool stability and usability, (typically the older and larger tools have an advantage here), but also flexibility in carrying out changes due to user requests (typically the younger and smaller tools have a higher degree of freedom).

Year. The year when development was started.

Tool	License	Year	Ver.	Code size
ATL/EMFTVM	EPL	2011[2]	3.3.0	50k Java
Epsilon	EPL	2005	0.9.1	397k Java
Edapt	EPL	2007		36k Java, 31k comment
GReTL	GPL	2009		28k Java, 19k comment
GrGen.NET	LGPL	2003	3.1	78k Java, 65k C#
GROOVE	Apache	2003	4.6.0	133k Java, 63k comment
Henshin	EPL	2009	EIP	57k Java, 41k comment
MDELab SDI	EPL	2008	2.3.5	104k Java, 49k comment
metatools	closed/money[1]	2001		103k Java
MOLA	closed/free	2005	1.3.5	82k C++
QVTR-XSLT	closed/free	2009		7.5k XSLT, 1.2k Java, 10k XML
UML-RSDS	open/free	1996	1.3	75k Java
VIATRA2	EPL	2002	3.2	144k Java, 104k comment

[1] academic licenses available
[2] the values are given for the EMFTVM extension, ATL is older

Table 23: License and Maturity

Version. The version number reached. EIP instead of a version number stands for Eclipse Incubation Project.

Code size. The lines of code written by programming language, according to cloc.sourceforge.net (excluding third-party libraries and examples).

7.5. Validation and Verification

"How can I ensure my program does what it should do?" is the primary question we answer in Table 24, extended by answers to the question "How can I use the tool for verification?". The helpers for manual inspection (graph viewer, debugger) were already compared in subsubsection 7.3.1, here we ask for the support for automatic checking. We begin by comparing the support for checking model integrity.

Constraints. Our classification for the support of metamodel or type graph constraints is combined from two groups. The first is the fact to be checked, being one of:

A: only certain attributes are allowed (depending on the type).

T: only certain edge types are allowed between certain node types.

M: only certain multiplicities of incoming/outgoing edges of a given type are allowed at specific nodes.

The second is the strictness of the check, being one of:

F: this is enforced.

C: this can be checked.

Facts of type checked can be violated during the transformation, this eases transformation writing. On the other hand it is easier to introduce errors as one must not forget to trigger the checking. The enforced checks may be in fact not checks at all, but consequences of the chosen model implementation: for attributes F is typically a consequence of model kind fixed as defined in Table 16), commonly implemented by statically typed classes containing the attributes as member variables. For types and multiplicities F is a hint at model-based tools, which implement edges by reference or container variables, whereas graph tools typically allow implementation-wise an arbitrary number of edges at each node. In case checks like these are not provided, they can be of course programmed.

Validation and Verification. Besides the basic checks for model integrity, OCL expressions may be used to validate the model; describing and checking constraints on object structures is in fact the primary reason for their existence. The tools may even offer a dedicated validation language extending OCL with e.g., refined user feedback. Furthermore, the tools may offer to clone a graph an rewrite it until an answer regarding a very complex constraint is reached.

Finally, the tool may be based on a correct-by-design approach: the user is not writing code to be executed, but describes precondition and postcondition formulas, which are then implemented by the code generator of the tool. Moreover, the tool may allow for formal verification of its transformations, with theorem proving support.

Until now we spoke of validating or verifying the user transformations. A task some of the tools introduced here were created for was verifying other systems. To this end, they offer model checking by state space enumeration (SSE). A state space enumerator allows to enumerate a state space of graphs generated by applying a rule set, visualizing the space structure as well as the single states. Unfolding the space of different executions is a help in manual inspection, too. But its primary use is in automated model checking: it is tested whether temporal logic formulas hold for all the enumerated states.

Tool	**Constraints**	**Validation and Verification**
ATL/EMFTVM	A:F, T:F, M:C	
Epsilon	A:F, T:F, M:F [1]	dedicated validation language
Edapt	T:C, M:C	
GReTL	A:F, T:F, M:C	
GrGen.NET	A:F, T:C, M:C	rewrite-clone, programmed SSE
GROOVE	A:F, T:F, M:C+F	SSE for model checking
Henshin	T:C+F, M:C+F	SSE for model checking
MDELab SDI	T:C+F, A:C, M:C	
metatools	A:F, T:F, M:F	
MOLA	A:F, T:F, M:F	
QVTR-XSLT	A:C, T:C	OCL checking
UML-RSDS	A:F, M:F, T:F	correct-by-design, export to theorem-prover
VIATRA2	T:C, M:C	stochastic SSE

[1] depends on the underlying modeling technology, values given for EMF

Table 24: Validation and verification support

8. Discussion and Related Work

The goal of the Hello World case was a comparison of the tools regarding their performance to handle simple task. Another case with the goal of basic tool comparison, a de-facto Hello World case for model transformation up to now was given in the invitation to the Model Transformations in Practice Workshop [5], with a Class-to-RDBMS scenario. Classes containing attributes linked by associations in between them were to be transformed to tables containing columns with foreign key constraints in between them. In contrast to the Hello World case that consists of several very simple tasks, the Class-to-RDBMS case consists of one task, which is far less simple, e.g., requiring support to handle structures extending into depth.

In Model Transformations by Graph Transformation [73], the graph transformation languages AGG, TGG, VIATRA, and VMTS are compared using the aforementioned object-to-relational transformation example. In the Comparison of Three Model Transformation Languages [21], the transformation languages CGT, AGG, and ATL are compared alongside a complex refactoring example.

Other transformation tool contest based comparisons. While even powerful tools and languages should offer simple solutions for simple tasks, a Hello World case as used for this tool comparison is less well-suited to assess tools regarding their ability to tackle complex cases. So it was complemented by a Program Understanding and a Compiler Optimization case (and a Model Migration, but this one received only two submissions), which evaluated how well the tools are suited for complex tasks involving large workloads.

The Program understanding case [32] required to extract a state machine model out of an abstract syntax graph of a computer program, according to some patterns in the ASG. A prerequisite was the ability to import XMI. The case was designed to measure the ability of tools to capture non-local data; it required to follow chains of potentially unbounded length. So declarative depth support as given in Table 6 was of high importance to achieve good results here, as underlined by the voting results, with GReTL scoring first, GrGen.NET second, and MDELab SDI third.

The Compiler optimization case [8] required to carry out constant folding (which is trivial for data flow and complex for control flow) and instruction selection on a graph-based compiler intermediate representation. A prerequisite was the ability to import GXL. The case was designed to measure the performance of the competing tools. Achieving a good result was aided by the ability to store visited nodes in containers as given in Table 5, so one could fold along the flow. So performance-oriented tools, which were able to handle large data sets achieved good results here, as underlined by the voting results with GrGen.NET scoring first, GReTL second, and GROOVE third.

Some cases of earlier editions of the Transformation Tool Contest have also resulted in the publication of tool comparisons: In [72] multiple tools are compared regarding their performance in creating Sierpiński triangles. In [79] multiple tools are compared regarding a transformation from UML activity diagrams to formal CSP processes. In [66] an empirical comparison of nine tools is given regarding the migration of UML activity diagrams from version 1.4 to version 2.2.

Tool comparison frameworks. Besides the tool comparisons based on the Transformation Tool Contest, other attempts at classifying transformation tools, tasks, and solutions need to be mentioned. Two are standing out, for one the Feature-based survey of model transformation approaches [10], which presents a classification model for transformation languages based on their technical properties. This paper is similar in the use of a huge amount of features, but it differs in the features chosen (based on the set of competing tools and the lessons learned from the cases), and it is surpassing the mentioned paper in the direct in-place application of the features to a large amount of tools. And for the other the Taxonomy of Model Transformations [57], which introduces different applications of transformation languages and tools, and presents functional as well as non-functional requirements for transformation approaches. In the Comparison of Taxonomies for Model Transformation Languages [74] the two aforementioned taxonomies are compared with two further classification schemes in a meta-comparison. The paper Benchmarking for Graph Transformation [82] finally proposes a benchmark for comparing the performance of graph transformation languages.

8.1. Review process support

As already introduced when describing the workflow, all tools were required to offer a SHARE image, which was to be investigated by the opponents.

Primary function. This review of the solution and tool together with the paper allows for much more honest results than paper only solutions. A paper can be tweaked easily to show a compelling solution by emphasizing the strong points of the solution and leaving out the dark corners entirely. An executable environment cannot be tweaked like this. Especially not with opponents, which explicitly want to find negative aspects so the competing tool can be voted down, and their own tool look correspondingly better.

SHARE was used massively for this purpose, as can be seen by the comments the opponents gave [8], e.g.: “Unfortunately, only the first task (...) is reproducible in the SHARE demo, the other solutions fail for some reason (...). Please fix this.”

Besides the SHARE hosting computer was used as the base for performance comparisons in the complex cases; this made the original numbers comparable, which were typically measured at wildly different machines.

Secondary function. The aim of the Transformation Tool Contest is to compare the participating tools, and to allow the tool vendors to learn from each other. The SHARE images are helpful in this regard, too, because they allow to inspect competing tools with a minimum of effort.

A case submitter that first only sent an archive file was asked by another participant for a SHARE demo: “(...) a SHARE demo really should

[8] http://planet-research20.org/ttc2011/index.php?option=com_community&view=groups&task=viewgroup&groupid=13&Itemid=150

be created for this solution. I just did not figure out how to operate the Java program in your submitted zip file." With SHARE, the reviewers can concentrate on reviewing the tool already set up, and prospective users on assessing the tool – they do not need to invest their time into getting the tool running (this effort is only needed for the tools the user finally chooses).

9. Conclusion and Outlook

"The primary goal of this article is to help software engineers that are facing a model transformation or graph rewriting problem in choosing a tool that is well-suited to their needs." So we began this article. A prospective tool user was then taken by the hand and lead through the maze of the numerous available tools: First, (s)he was introduced into the world of transformation tools, with explanations of core notions and an overview of the position of the tools in the tool landscape. Then, calling cards of the tools could be inspected, displaying an illustrative example solution of one of the Hello World tasks, and discussing the tasks the tools are suited for. The set of candidates that was resulting could be ordered and reduced further by paying attention to the votes given for the solutions at the Transformation Tool Contest, and especially by inspecting detailed feature matrices. Finally, it was possible to condense the remaining set of candidates into the most promising tool by an in depth evaluation of the SHARE images of the solutions to the tasks posed at the Transformation Tool Contest.

In contrast to most other tool comparisons, this one was focused on potential requirements of users, taking factors like prerequisites for execution, license, and user documentation into account. And in contrast to all other previous tool comparisons we were able to evaluate a large number of tools, based on the impressive number of submissions to the Hello World case posed at the Transformation Tool Contest 2011. The classification was applied to many but not all of the state-of-the-art tools. A worthwhile endeavor would be an extension with the missing tools yielding a full market survey. Other aspects of tool comparison were unfortunately beyond the scope of this article centered around the tool basics and computational expressiveness, especially to mention are tool performance and scalability to large tasks.

The article gave an example for software engineering in the cloud, which is an application of cloud computing for the benefit of software engineering, as opposed to the older topic of software engineering for the cloud, which is concerned with building applications for the cloud. It was explained how

SHARE was employed in order to achieve strongly validated and reproducible results. While fully satisfying the reviewing requirements of scientific events, even improving on the state-of-the-art, we want to note that the anonymous availability of SHARE images to the public would further increase its usefulness: prospective tool users from industry could then evaluate their tools of choice without writing personally to the SHARE maintainer(s).

Acknowledgments. We want to thank further authors of this article: the ones which do not appear in the title but have contributed:

References

[1] Arendt, T., Bierman, E., Jurack, S., Krause, C., Taentzer, G., 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. MoDELS'10. LNCS 6394. Springer, pp. 121–135, dOI: 10.1007/978-3-642-16145-2_9.

[2] Barmpis, K., Kolovos, D., 2012. Comparative analysis of data persistence technologies for large-scale models. In: Extreme Modeling Workshop, ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems.

[3] Bedaride, P., Gardent, C., 2009. Semantic normalisation: a framework and an experiment. In: Proceedings of the Eighth International Conference on Computational Semantics. IWCS-8 '09. Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 359–370.

[4] Bergmann, G., Ráth, I., Varró, G., Varró, D., 2012. Change-driven model transformations. change (in) the rule to rule the change. Software and Systems Modeling 11, 431–461.

[5] Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., 2005. Model transformations in practice workshop. Call for papers.
URL http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf

[6] Blomer, J., Geiß, R., Jakumeit, E., Jul. 2011. The GrGen.NET User Manual. http://www.grgen.net.

[7] Börger, E., Stärk, R., 2003. Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag.

[8] Buchwald, S., Jakumeit, E., 2011. Compiler optimization: A case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 6–16.

[9] Clowes, D., Kolovos, D. S., Holmes, C., Rose, L. M., Paige, R. F., Johnson, J., Dawson, R., Probets, S. G., 2010. A reflective approach to model-driven web engineering. In: ECMFA. Vol. 6138 of Lecture Notes in Computer Science. Springer, pp. 62–73.

[10] Czarnecki, K., Helsen, S., Jul. 2006. Feature-based survey of model transformation approaches. IBM Systems Journal 45 (3), 621–645. URL http://dx.doi.org/10.1147/sj.453.0621

[11] Dan, L., 2011. SHARE image of the QVTR XSLT solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_TTC11_QVTR-XSLT.vdi

[12] Ebert, J., Bildhauer, D., 2010. Reverse engineering using graph queries. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (Eds.), Graph Transformations and Model-Driven Engineering. Vol. 5765 of Lecture Notes in Computer Science. Springer, pp. 335–362.

[13] Ebert, J., Horn, T., 2012. GReTL: an extensible, operational, graph-based transformation language. Software & Systems Modeling, 1–21. URL http://dx.doi.org/10.1007/s10270-012-0250-3

[14] Edgar Jakumeit, S. B., 2011. SHARE image of the GrGen.NET solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_GrGen_v2.vdi

[15] Fischer, T., Niere, J., Torunski, L., Zündorf, A., Nov. 2000. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations. Vol. 1764/2000 of LNCS. Springer-Verlag, London, UK, pp. 296–309.

[16] Gabrysiak, G., Giese, H., Seibel, A., Sep. 2010. Deriving behavior of multi-user processes from interactive requirements validation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE'10. ACM, Antwerp, Belgium, pp. 355–356.

[17] Galvao Lourenco da Silva, I., Zambon, E., Rensink, A., Wevers, L., Akşit, M., 2011. Knowledge-based graph exploration analysis. In: Schürr, A., Varró, D., Varró, G. (Eds.), Fourth International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE). Vol. 7233 of Lecture Notes in Computer Science. pp. 105–120.

[18] Ghamarian, A. H., de Mol, M. J., Rensink, A., Zambon, E., Zimakova, M. V., 2012. Modelling and analysis using groove. International journal on software tools for technology transfer 14 (1), 15–40.

[19] Giese, H., Hildebrandt, S., Seibel, A., 2009. Improved Flexibility and Scalability by Interpreting Story Diagrams. In: Magaria, T., Padberg, J., Taentzer, G. (Eds.), Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009). Vol. 18. Electronic Communications of the EASST.

[20] Gorp, P. V., Mazanek, S., 2011. Share: a web portal for creating and sharing executable research papers. Procedia CS 4, 589–597.

[21] Grønmo, R., Møller-Pedersen, B., Olsen, G., 2009. Comparison of three model transformation languages. In: ECMDA-FA'09: European Conference on Model Driven Architecture - Foundations and Applications. Vol. 5562 of LNCS. Springer, pp. 2–17.

[22] Hegedüs, Á., 2011. SHARE image of the Viatra2 solution.
URL `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_VIATRA.vdi`

[23] Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D., 11/2011 2011. A model-driven framework for guided design space exploration. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, Lawrence, Kansas, USA.

[24] Hegedüs, Á., Ujhelyi, Z., Ráth, I., Horváth, Á., 2011. Visualization of traceability models with domain-specific layouting. Electronic Communications of the EASST, Proceedings of the Fourth International Workshop on Graph-Based Tools 32.

[25] Helms, B., Shea, K., Hoisl, F., 2009. A framework for computational design synthesis based on graph-grammars and function-behavior-structure. ASME Conference Proceedings 2009 (49057), 841–851.

[26] Herrmannsdoerfer, M., 2011. Gmf: A model migration case for the transformation tool contest. In: TTC. pp. 1–5.

[27] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2008. Automatability of coupled evolution of metamodels and models in practice. In: MoDELS '08.

[28] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2008. COPE: A language for the coupled evolution of metamodels and models. In: MCCM '08.

[29] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2009. COPE - automating coupled evolution of metamodels and models. In: ECOOP '09.

[30] Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G., 2010. An extensive catalog of operators for the coupled evolution of metamodels and models. In: SLE '10.

[31] Herrmansdörfer, M., 2011. SHARE image of the Edapt solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_EMFEdapt.vdi

[32] Horn, T., 2011. Program understanding: A reengineering case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 17–21.

[33] Horn, T., 2011. SHARE image of the GReTL solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi

[34] Horn, T., Ebert, J., 2011. The GReTL Transformation Language. In: Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. Vol. 6707 of Lecture Notes in Computer Science. Springer, pp. 183–197.

[35] Horváth, Á., Bergmann, G., Ráth, I., Varró, D., 2010. Experimental assessment of combining pattern matching strategies with VIATRA2. International Journal on Software Tools for Technology Transfer (STTT) 12, 211–230.

[36] Interdisciplinary Centre for Security, Reliability and Trust (SnT), 2011. Annual report 2011. URL http://www.uni.lu/content/download/52106/624943/version/1/file/SnT_AR2011_final_web.pdf, 14–15.

[37] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Jun. 2008. ATL: A model transformation tool. Science of Computer Programming 72 (1-2), 31–39.

[38] Jurack, S., Tietje, J., 2011. Saying hello world with Henshin - a solution to the TTC 2011 instructive case. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 253–280, dOI: 10.4204/EPTCS.74.22.

[39] Kalnina, E., 2011. SHARE image of the Mola solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_MOLA.vdi

[40] Kalnins, A., Barzdins, J., Celms, E., 2004. Model transformation language MOLA. In: Proceedings of MDAFA 2004. Linkoeping, Sweden, pp. 14–28.

[41] Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J., 2007. Building tools by model transformations in Eclipse. In: Proceedings of DSM'07 Workshop of OOPSLA 2007. Jyvaskyla University Printing House, Montreal, Canada, pp. 194–207.

[42] Kastenberg, H., Rensink, A., 2006. Model checking dynamic states in groove. In: Valmari, A. (Ed.), Model Checking Software (SPIN), Vienna,

Austri. Vol. 3925 of Lecture Notes in Computer Science. Springer-Verlag, pp. 299–305.

[43] Kolovos, D., Paige, R., Polack, F., 2006. The Epsilon Object Language (EOL). In: ECMDA-FA. Vol. 4066 of Lecture Notes in Computer Science. Springer, pp. 128–142.

[44] Krause, C., 2011. SHARE image of the Henshin solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_Henshin_v2.vdi

[45] Krause, C., Giese, H., 2012. Probabilistic graph transformation systems. In: ICGT'12 – International Conference on Graph Transformation. Vol. 7562 of Lecture Notes in Computer Science. Springer-Verlag, pp. 311–325.

[46] Kurtev, I., Bézivin, J., Aksit, M., 2002. Technological spaces: An initial appraisal. In: CoopIS, DOA Federated Conferences, Industrial track.

[47] Lano, K., 2011. SHARE image of the UML RSDS.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC10_uml-rsds-livecontest_TTC11.vdi

[48] Lano, K., Kolahdouz-Rahimi, S., 2011. Specification of the gmf migration case study. In: Transformation Tool Contest 2011.

[49] Lano, K., Kolahdouz-Rahimi, S., 2012. Constraint-based specification of model transformations. Journal of Systems and Software, to appear.

[50] Lano, K., Kolahdouz-Rahimi, S., Clark, T., 2012. Comparing model transformation verification approaches. In: Modevva workshop, MODELS 2012.

[51] Lepper, M., 2011. SHARE image of the metatools solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_metatools.vdi

[52] Lepper, M., Trancón y Widemann, B., 2011. Optimization of visitor performance by reflection-based analysis. In: Proceedings of the 4th international conference on Theory and practice of model transformations. ICMT'11. Springer-Verlag, Berlin, Heidelberg, pp. 15–30.

[53] Lepper, M., Trancón y Widemann, B., 2011. Solving the TTC 2011 compiler optimization task with metatools. In: Gorp, P. V., Mazanek, S., Rose, L. (Eds.), TTC. Vol. 74 of EPTCS. pp. 70–115.

[54] Liepiņš, R., 2011. lQuery: A model query and transformation library. Scientific Papers, University of Latvia 770, 27–45.

[55] Mazanek, S., 2011. Helloworld! an instructive case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 22–26, available at http://sites.google.com/site/helloworldcase/.

[56] Mazanek, S., Rutetzki, C., Minas, M., 2011. Tool demonstration of the transformation judge. In: AGTIVE 2011. Springer, accepted for Publication.

[57] Mens, T., Gorp, P. V., 2006. A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142.

[58] Miller, J., Mukerji, J. (Eds.), 2003. MDA Guide. Object Management Group.

[59] MOLA team, 2011. Mola pages.
URL http://mola.mii.lu.lv

[60] Ráth, I., Bergmann, G., Ökrös, A., Varró, D., 2008. Live model transformations driven by incremental pattern matching. In: Vallecillo, A., Gray, J., Pierantonio, A. (Eds.), Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT 2008). Vol. 5063/2008 of LNCS. Springer Berlin / Heidelberg, p. 107–121.

[61] Ráth, I., Vágó, D., Varró, D., 2008. Design-time simulation of domain-specific models by incremental pattern matching. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008,

Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings. IEEE, pp. 219–222.

[62] Rensink, A., 2011. SHARE image of the GROOVE solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_groove-helloworld.vdi

[63] Rensink, A., Heckel, R., König, B., Jul. 2007. Graph transformation for concurrency and verification - preface. In: Proceedings of the Workshop on Graph Transformation for Concurrency and Verification (GT-VC). Vol. 175 of Electronic Notes in Theoretical Computer Science 175. Elsevier, Amsterdam, pp. 1–2.
URL http://doc.utwente.nl/61779/

[64] Rose, L., 2011. SHARE image of the Epsilon solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_epsilon-helloworld.vdi

[65] Rose, L., Kolovos, D., Paige, R., Polack, F., 2010. Model migration with Epsilon Flock. In: ICMT. Vol. 6142 of Lecture Notes in Computer Science. Springer, pp. 184–198.

[66] Rose, L. M., Herrmannsdörfer, M., Mazanek, S., Gorp, P. V., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., Wimmer, M., 2012. Graph and model transformation tools for model migration. Journal on Software and Systems ModelingAccepted; to appear.

[67] Rose, L. M., Paige, R. F., Kolovos, D. S., Polack, F., 2008. The epsilon generation language. In: Schieferdecker, I., Hartman, A. (Eds.), ECMDA-FA. Vol. 5095 of Lecture Notes in Computer Science. Springer, pp. 1–16.

[68] Seidewitz, E., Sep. 2003. What models mean. IEEE Softw. 20 (5), 26–32.
URL http://dx.doi.org/10.1109/MS.2003.1231147

[69] Śmiałek, M., Kalnins, A., Kalnina, E., Ambroziewicz, A., Straszak, T., Wolter, K., 2010. Comprehensive system for systematic case-driven software reuse. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorn'y, J.,

Rumpe, B. (Eds.), SOFSEM 2010: Theory and Practice of Computer Science. Vol. 5901 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 697–708.

[70] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF: Eclipse Modeling Framework 2.0, 2nd Edition. Addison-Wesley Professional.

[71] Taentzer, G., 2004. AGG: A graph transformation environment for modeling and validation of software. In: Proc. AGTIVE'03. LNCS 3062. Springer, pp. 446–453, dOI: 10.1007/b98116.

[72] Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T., 2007. Generation of sierpinski triangles: A case study for graph transformation tools. In: AGTIVE. pp. 514–539.

[73] Taentzer, G., Ehrig, K., Guerra, E., Lara, J. D., Levendovszky, T., Prange, U., Varro, D., 2005. Model transformations by graph transformations: A comparative study. In: Model Transformations in Practice Workshop at MODELS 2005, Montego. p. 05.

[74] Tamura, G., Cleve, A., Mar. 2010. A Comparison of Taxonomies for Model Transformation Languages. Paradigma 4 (1), 1–14.

[75] Trancón y Widemann, B., Lepper, M., 2012. Paisley: Pattern matching à la carte. In: Theory and Practice of Model Transformations. Vol. 7307 of LNCS. Springer.

[76] Trancón y Widemann, B., Lepper, M., Wieland, J., 2003. Automatic construction of XML-based tools seen as meta-programming. Automated Software Engineering 10, 23–38, 10.1023/A:1021864801049.
URL http://dx.doi.org/10.1023/A:1021864801049

[77] Ujhelyi, Z., Horváth, Á., Varró, D., 2012. Dynamic backward slicing of model transformations. In: International Conference on Software Testing and Validation, 2012. IEEE, Montreal, Canada.

[78] Van Gorp, P., Mazanek, S., Rose, L. (Eds.), 2011. TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings. EPTCS.

[79] Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E., Weinell, E., 2008. Transformation of UML models to CSP: A case study for graph transformation tools. In: AGITIVE'08: International Symposium on Applications of Graph Transformation with Industrial Relevance. Vol. 5088 of LNCS. Springer, pp. 540–565.

[80] Varró, D., Balogh, A., October 2007. The model transformation language of the VIATRA2 framework. Science of Computer Programming 68 (3), 214–234.

[81] Varró, D., Pataricza, A., October 2003. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling 2 (3), 187–210.

[82] Varró, G., Schürr, A., Varró, D., 2005. Benchmarking for graph transformation. In: VL/HCC. pp. 79–88.

[83] VIATRA2 Framework:An Eclipse GMT Subproject, 2012.
URL http://viatra.inf.mit.bme.hu

[84] von Detten, M., Heinzemann, C., Platenius, M., Rieke, J., Travkin, D., Hildebrandt, S., 2012. Story Diagrams - Syntax and Semantics. Tech. rep., Software Engineering Group, Heinz Nixdorf Institute, version 0.2.

[85] Wagelaar, D., 2011. SHARE image of the EMFTVM solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_livecontest-dennis_EMFTVM-HelloWorld.vdi

[86] Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A., May 2012. Translational semantics of a co-evolution specific language with the emf transformation virtual machine. In: Proceedings of ICMT 2012. Vol. 7307 of Lecture Notes in Computer Science. Springer-Verlag, pp. 192–207.

[87] Wagelaar, D., Tisi, M., Cabot, J., Jouault, F., Oct. 2011. Towards a general composition semantics for rule-based model transformation. In: Proceedings of MoDELS 2011. Vol. 6981 of Lecture Notes in Computer Science. Springer-Verlag, pp. 623–637.

[88] Warmer, J., Kleppe, A., 2003. The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[89] Wätzold, S., 2011. SHARE image of the MDE Lab solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_sdm-hpi.vdi